ESCUELA POLITÉCNICA SUPERIOR
DEPARTAMENTO DE INFORMÁTICA

UNIVERSIDAD CARLOS III DE MADRID
PROYECTO FIN DE CARRERA

INGENIERÍA EN INFORMÁTICA

"Gestión de SmartCards mediante PKCS#11"

AUTOR: IGNACIO ÁLVAREZ SANTIAGO

TUTOR: JOSÉ MARÍA SIERRA CÁMARA

Índice de contenido

1 – INTRODUCCIÓN 11
1.1 – INTRODUCCIÓN AL PROYECTO 11
1.2 – MOTIVACIÓN DEL PROYECTO 12
1.3 – ENFOQUE DEL PROYECTO 13
1.4 – OBJETIVOS DEL PROYECTO 14
1.5 – ESTRUCTURA DE LA MEMORIA 16
2 – TECNOLOGÍAS SOFTWARE 18
2.1 – PKCS#10 Y CERTIFICADOS 19
2.1.1 – ASN.1 19
2.1.2 – CODIFICACIÓN DE CERTIFICADOS Y CSR 24
2.1.3 – DETALLES DE LOS CERTIFICADOS 27
2.2 – PKI 29
2.3 – RSA 33
2.4 – PKCS#12 36
2.5 – PKCS#15 38
2.6 – PKCS#11 43
2.6.1 – SESIONES Y USUARIOS 44
2.6.2 – TIPOS DE DATOS PKCS#11 46
2.6.3 – OBJECTOS PKCS#11 49
2.6.4 – FUNCIONES PKCS#11 54
2.6.5 – UTILIZACIÓN DE PKCS#11 58
2.7 – OPENSSL 60
3 – SMARTCARDS 62
3.1 – CARACTERÍSTICAS TÉCNICAS DE LAS SMARTCARDS USADAS 63
3.1.1 – CERES 63
3.1.2 – STARCOS 63
3.1.3 – ALADDIN (TARJETA Y ETOKEN) 64
3.1.4 – DNIE 64
3.2 – COMPARATIVA DE LAS TARJETAS 65

4 – ANÁLISIS DEL PROYECTO 69
4.1 – CAPACIDADES DEL SISTEMA 70
4.2 – RESTRICCIONES DEL SISTEMA 72
4.2.3 – RESTRICCIONES HARDWARE DE LAS TARJETAS 72
4.3 – USUARIOS 74
4.4 – ENFOQUE DE LA SOLUCIÓN 75
4.4.1 – TIPO DE INTERFAZ 76
4.5 – CASOS DE USO 77
4.5.1 – DESCRIPCIÓN TEXTUAL DE LOS CASOS DE USO 81
5 – DISEÑO 92
5.1 – MÉTODO DE DISEÑO 93
5.2 – DESCRIPCIÓN DE LOS COMPONENTES 94
5.2.1 – Módulo Principal 97
5.2.1.1 – Funciones del Módulo Principal 98
5.2.1.2 – Interfaz textual de usuario 100
5.2.2 – Módulo PKCS#11 103
5.2.2.1 – Iniciar Sistema 105
5.2.2.2 – Finalizar Sistema 106
5.2.2.3 – Obtener número de Slot del Sistema 107
5.2.2.4 – Obtener Listado de Slots del Sistema 107
5.2.2.5 – Información de Slot 108
5.2.2.6 – Información Extendida de Slot 108
5.2.2.7 – Información del Token 108
5.2.2.8 – Abrir Sesión 109
5.2.2.9 – Cerrar Sesión 110
5.2.2.10 – Login 110
5.2.2.11 – Logout 111
5.2.2.12 – Cambio de PIN 112
5.2.2.13 – Recorrer Elementos 113
5.2.2.14 – Generar Números Aleatorios 114
5.2.2.15 – Crear Certificado 115
5.2.2.16 – Crear Clave Privada 117

5.2.2.17 – Crear Clave Pública 119
5.2.2.18 – Cambiar ID de objetos 120
5.2.2.19 – Cambiar etiqueta de objetos 121
5.2.2.20 – Obtener longitud de la clave pública 122
5.2.2.21 – Obtener longitud de la clave privada 123
5.2.2.22 – Eliminar Objetos 124
5.2.2.23 – Crear Par de Claves 125
5.2.2.24 – Firmar Ficheros 127
5.2.2.25 – Firmar Buffers 129
5.2.2.26 – Comprobar Firma de Ficheros 129
5.2.2.27 – Listar Mecanismos 129
5.2.2.28 – Listar Mecanismos 130
5.2.2.29 – Longitud de Certificado 132
5.2.2.30 – Obtener Certificado 133
5.2.2.31 – Mostrar Clave Pública 133
5.2.2.32 – Desbloquear PIN 134
5.2.3 – Módulo LISTA 135
5.2.3.1 Objeto Nuevo 137
5.2.3.2 Crear Lista 137
5.2.3.3 Reinicializar plantilla 137
5.2.3.4 Reinicializar plantilla 137
5.2.3.5 Recorrer Objetos 138
5.2.3.7 Elementos Tipo 139
5.2.3.8 Manejador Posición 139
5.2.3.9 Insertar 139
5.2.3.10 Borrar 140
5.2.4 – Módulo MECANISMOS 140
5.2.4.1 Tamaño Mecanismos 140
5.2.4.2 Tamaño Compatibles 141
5.2.4.3 Tamaño Compatibles 141
5.2.5 – Módulo SOPORTE 142
5.2.5.1 Mostrar Campo 142

5.2.5.2 Mostrar Campo Hexadecimal 143
5.2.5.3 Mostrar Clase 143
5.2.5.4 ¿Es Entero? 143
5.2.5.5 Valor del parámetro 143
5.2.5.6 Comparar Cadenas 144
5.2.5.7 Traducir Mecanismos 144
5.2.5.8 Traducir Error 144
5.2.5.9 Obtener Longitud 144
5.2.5.10 Invertir Bytes de Cadena 145
5.2.5.11 Asignar Entero a Cadena 145
5.2.5.12 Formato Fecha 145
5.2.5.13 Pausa 145
5.2.5.14 Limpiar 146
5.2.5.15 Corregir Cadena 146
5.2.5.16 Mostrar Lista de Lectores 146
5.2.6 Módulo OPENSSL 146
5.2.6.1 Crear Petición de Certificado 148
5.2.6.2 Leer Certificado 150
5.2.6.3 Leer PKCS#12 150
5.2.6.4 Exportar Certificado X.509 151
5.2.6.5 Mostrar Certificado X.509 152
6 – CONCLUSIONES 153
7 – FUTUROS DESARROLLOS 156
7.1 – IMPLEMENTACIÓN DE UNA GUI 156
7.2 – IMPLEMENTACIÓN ORIENTADA A OBJETOS 158
7.3 – LENGUAJES DE MARCADO 160
7.4 – SOPORTE PARA CA 162
8 – GLOSARIO 163
9 – BIBLIOGRAFÍA 168
APÉNDICES 170
ESTUDIO DEL DESARROLLO DEL PROYECTO 170
Descripción de las tareas 171

Análisis 171
Aprendizaje 171
Diseño 172
Implementación 172
Pruebas 173
Documentación 173
SOFTWARE UTILIZADO 175
SISTEMAS OPERATIVOS 175
Windows XP 175
Windows Server 2003 175
OTRO SOFTWARE 176
Cryptoki 176
OpenSSL 176
MinGW 177
NetBeans 177
Microsoft Word 178
Microsoft Excel 179
Microsoft Visio 180
Microsoft Project 181
Altova Umodel 182
VMWare Player 183
ASN.1 Editor 184
Notepad++ 185
OPENSSL 187
Fichero de configuración 187
Comandos para gestionar OpenSSL 189
PKCS#11 191
Estructuras usadas de PKCS#11 191
Flags de PKCS#11 193

Índice de ilustraciones

Ilustración 1: Estructura DER 20
Ilustración 2: Proceso de obtención de un certificado X.509 30
Ilustración 3: Cadenas de confianza PKI 32
Ilustración 4: Firmado y comprobación 34
Ilustración 5: PKCS#15 como capa intermedia 38
Ilustración 6: Primer nivel de PKCS#15 39
Ilustración 7: Nivel principal de PKCS#15 40
Ilustración 8: Nivel del EF(ODF) 40
Ilustración 9: Nivel de EF(CDF) – Certificados 41
Ilustración 10: Ejemplo de creación de un certificado 42
Ilustración 11: Sesiones en PKCS#11 45
Ilustración 12: Objetos PKCS#11 50
Ilustración 13: Tarjeta Ceres 63
Ilustración 14: Tarjeta Starcos 2.3 (Blank Card) 63
Ilustración 15: Aladdin eToken 64
Ilustración 16: DNIe 64
Ilustración 17: Tiempo de generación RSA - 1024 bits 66
Ilustración 18: Tiempo de generación RSA - 2048 bits 66
Ilustración 19: Tiempo de importación de un Certificado X.509 (2.3 Kb) 67
Ilustración 20: Tiempo de firma de un fichero de 1 Mb con una clave RSA de 1024 bits usando SHA1 67
Ilustración 21: Casos de uso del Usuario sin sesión 78
Ilustración 22: Casos de uso del Usuario Normal 79
Ilustración 23: Casos de uso del Oficial de seguridad 80
Ilustración 24: Diagrama de despliegue 95
Ilustración 25: Diagrama de componentes 96
Ilustración 26: Ejemplo función Módulo Principal "Firma de Fichero" 100
Ilustración 27: iniciarSistema 106
Ilustración 28: finalizarSistema 106
Ilustración 29: obtenerNumeroSlot 107
Ilustración 30: infoSlot 108
Ilustración 31: infoToken 109
Ilustración 32: abrirSesion 109
Ilustración 33: cerrarSession 110
Ilustración 34: login 111
Ilustración 35: logout 112
Ilustración 36: CambioPIN 112
Ilustración 37: recorrerElementos 113

Ilustración 38: generarNumeroAleatorio 115
Ilustración 39: crearCertificado 116
Ilustración 40: cambioID 120
Ilustración 41: cambioEtiqueta 122
Ilustración 42: obtenerLongitudClavePublica 123
Ilustración 43: obtenerlongitudClavePrivada 124
Ilustración 44: destruirObjeto 125
Ilustración 45: crearParDeClaves 125
Ilustración 46: firmaFichero 128
Ilustración 47: mecanismos 130
Ilustración 48: valorClavePublica 131
Ilustración 49: longitudCertificado 132
Ilustración 50: mostrarClavePublica 133
Ilustración 51: iniciarPIN 134
Ilustración 52: Lista de objetos 135
Ilustración 53: GUI aplicación 157
Ilustración 54: Esbozo de un diagrama de clases 159
Ilustración 55: Fases del proyecto 170
Ilustración 56: Gantt del proyecto 174
Ilustración 57: Pantalla de NetBeans 178
Ilustración 58: Pantalla de Microsoft Word 179
Ilustración 59: Pantalla de Microsoft Excel 180
Ilustración 60: Pantalla de Microsoft Visio 181
Ilustración 61: Pantalla de Project 182
Ilustración 62: Pantalla de Altova Umodel 183
Ilustración 63: Pantalla de VMWare Player 184
Ilustración 64: Pantalla de ASN.1 editor 185
Ilustración 65: Pantalla de Notepad++ 186

Índice de tablas

Tabla 1: Tipos DER básicos 21
Tabla 2: Codificación Base 64 26
Tabla 3: Sesiones PKCS#11 46
Tabla 4: Prefijos de PKCS#11 47
Tabla 5: Tipos de datos PKCS#11 48
Tabla 6: Atributos de los objetos PKCS#11 53
Tabla 7: Descripción de las funciones PKCS#11 57
Tabla 8: Características Ceres 63
Tabla 9: Características Starcos 63
Tabla 10: Características Aladdin 64
Tabla 11: Características DNIe 64
Tabla 12: Caso de Uso “Salir de la aplicación” 81
Tabla 13: Caso de Uso “Login” 82
Tabla 14: Caso de Uso “Mostrar Certificados” 82
Tabla 15: Caso de Uso “Listar Objetos Públicos” 82
Tabla 16: Casos de Uso “Listar Mecanismos” 83
Tabla 17: Caso de Uso “Ver Información del Lector” 83
Tabla 18: Caso de Uso “Ver información de la Tarjeta” 83
Tabla 19: Caso de Uso “Mostrar Clave Pública” 84
Tabla 20: Caso de Uso “Exportar Certificado” 84
Tabla 21: Caso de Uso “Generar Números Aleatorios” 84
Tabla 22: Caso de Uso “Comprobar Firma de Fichero” 85
Tabla 23: Caso de Uso “Crear CSR” 85
Tabla 24: Caso de Uso "Listar todos los Objetos" 86
Tabla 25: Caso de Uso "Cambio de PIN de Usuario Normal" 86
Tabla 26: Caso de Uso "Cambio de Etiqueta - General" 86
Tabla 27: Caso de Uso "Cambio de ID - General" 87
Tabla 28: Caso de Uso "Firmar Fichero" 87
Tabla 29: Caso de Uso "Borrar Objetos - General" 88
Tabla 30: Caso de Uso "Generar Par de Claves" 88
Tabla 31: Caso de Uso "Importar PKCS#12" 89
Tabla 32: Caso de Uso "Importar Certificado" 89
Tabla 33: Caso de Uso "Logout Usuario Normal" 89
Tabla 34: Caso de Uso "Cambio de ID - Públicos" 90
Tabla 35: Caso de Uso “Cambio de Etiqueta - Públicas” 90
Tabla 36: Caso de Uso "Borrar Objetos - Públicos" 90
Tabla 37: Caso de Uso "Cambiar PIN Oficial de Seguridad" 91
Tabla 38: Caso de Uso "Logout de Oficial de Seguridad" 91

Tabla 39: Caso de Uso "Desbloquear PIN de Usuario Normal" 91
Tabla 40: Relaciones entre Componentes 97
Tabla 41: Información del lector 193
Tabla 42: información de la tarjeta 194
Tabla 43: Información de la sesión 194
Tabla 44: información de los mecanismos 194

1 – INTRODUCCIÓN

1.1 – INTRODUCCIÓN AL PROYECTO

El presente proyecto consiste en la generación de una herramienta capaz de gestionar Tarjetas Inteligentes (SmartCards) mediante el uso del estándar PKCS#11. Para ello, se ha usado el lenguaje de programación C, el Sistema Operativo Windows (aunque la finalidad es que el código fuente sea funcional en cualquier entorno) y las librerías PKCS#11 de los distintos fabricantes de las tarjetas probadas.

Actualmente, debido a las diferencias entre las distintas tarjetas, es muy difícil encontrar aplicaciones que permitan gestionarlas de forma transparente al usuario. Las que existen, que son en general los navegadores Web como Mozilla Firefox, permiten un uso básico de la tarjeta, limitado a la visualización de certificados, su importación y la eliminación de los mismo, además de su uso interno como token de firmado para la autenticación. El problema es que esto deja muy poca capacidad de utilización de la tarjeta al usuario. También existen las distintas aplicaciones propias de las tarjetas, pero además de que sólo son válidos para su modelo de tarjeta concreto, tienen, al igual que los navegadores, unas funcionalidades limitadas, más enfocadas a poder usar la tarjeta en ciertos escenarios que a facilitar una capa de funcionalidad extendida que el propio usuario pueda utilizar.

Por todo lo mencionado, este proyecto resulta especialmente interesante, ya que permitirá la gestión de todo tipo de Tarjetas Inteligentes que soporten el estándar PKCS#11, de modo que se puedan explotar las funcionalidades más interesantes que éste ofrece. Además, cambiando de DLL (en el proceso de compilación del programa) se obtendrían versiones usables con nuevos tipos de tarjetas, de manera que mientras no se produjesen cambios profundos en el estándar, cosa que sería muy improbable, la aplicación seguiría funcionando con los nuevos modelos de tarjetas que fueran apareciendo siempre que fueran acompañadas de sus correspondientes librerías PKCS#11.

1.2 – MOTIVACIÓN DEL PROYECTO

Como ya se ha comentado en la introducción al presente proyecto, el espíritu del mismo es poder generar una aplicación que dé soporte para la gestión de Tarjetas Inteligentes que implementen PKCS#11, dando un soporte extendido de su funcionalidad y que se pueda cambiar de tipo de tarjeta de forma tan sencilla como utilizar la nueva DLL de la nueva tarjeta.

Las principales aplicaciones, hoy en día, que hacen uso de módulos PKCS#11 para la gestión de las tarjetas, son los navegadores Web. Esto se debe a que el estándar criptográfico PKCS#11 está enfocado a integrarse en arquitecturas PKI, por lo que el usar tarjetas que contienen certificados (elemento básico de PKI) permite al navegador autenticar al usuario ante terceros, como por ejemplo, un servidor Web que requiera de un certificado del cliente.

El problema de los navegadores, es que el uso que le permiten al usuario de la tarjeta está orientado a la introducción y eliminación de certificados, por lo que cualquier uso más concreto, no relacionado con la autenticación Web, no es tratado por los navegadores.

Se podría pensar que bastaría con usar las propias aplicaciones software suministradas por los fabricantes de las tarjetas para así obtener un mayor control sobre las tarjetas, pero esto plantea básicamente dos problemas. En primer lugar, es que muy pocos fabricantes suministran software para distintos sistemas operativos (en la práctica casi ninguno ofrece software más allá de Windows). En segundo lugar, es que estas aplicaciones están enfocadas a la gestión de los objetos de la tarjeta. Esto, en la práctica, supone que dichas aplicaciones sólo permiten la inserción y eliminación de certificados o ficheros PKCS#12.

Por el contrario, el presente proyecto pretende que el usuario pueda realizar un uso más concreto de la tarjeta, permitiéndole no sólo visualizaciones de los elementos que contiene o su eliminación, sino su creación, gestión y lo que es más importante, permitir al usuario realizar firmas electrónicas en un ambiente local, enfocadas a su uso particular y no al entorno de la Web.

1.3 – ENFOQUE DEL PROYECTO

La idea de la realización de este proyecto era obtener una aplicación que permitiera gestionar tokens criptográficos bajo el estándar PKCS#11 de modo que se pudiera adaptar, de la mejor forma posible, a las distintas tarjetas. En principio, sólo sería necesario cambiar de módulo PKCS#11 (propio de cada tarjeta) para gestionar nuevos tokens. Pero eso, es sólo en la teoría. En la práctica, hay que hacer pequeños retoques sobre el código para asegurar la completa compatibilidad de éste ante nuevas tarjetas. Esto se debe a que todas las SmartCards tienen una interfaz PKCS#11 para que el desarrollador pueda trabajar con ellas, pero la forma en que dicha interfaz PCS#11 trabaja con las tarjetas es diferente. Esto se debe a la organización interna de la información en las tarjetas, lo que se engloba en otro estándar criptográfico denominado PKCS#15.

Por tanto, para dar una mejor visión al proyecto, hubo que trabajar con una tipología extendida de tarjetas, para así poder comprobar esas pequeñas diferencias y adaptar el código para hacerlo lo más genérico posible. Esto permite garantizar, de mejor forma, que el código se adaptará de la mejor manera posible a nuevos módulos PKCS#11; pero, aun así, en muchas ocasiones se requiere el uso de directivas de preprocesamiento para hacer un código ad hoc para un tipo de tarjeta en cuestión. Así que, aunque la finalidad fuera compatibilidad universal, para garantizar compatibilidad total con un tipo de tarjeta nuevo se debe estudiar el comportamiento de la tarjeta ante el estándar para así poder hacer los ajustes precisos.

El proyecto da como resultado una aplicación capaz de gestionar SmartCards por medio de PKCS#11. El código de dicha aplicación sería fácilmente reutilizable en otros proyectos/aplicaciones que tuvieran relación con el manejado de SmartCards, ya que una idea fundamental es obtener una codificación los más reutilizable posible.

Por poner un ejemplo, si quisiéramos hacer que el sistema operativo Windows (en versiones XP o 2003) utilizase la tarjeta para realizar operaciones como las de logon, tendría que ser capaz de hablar con la tarjeta. Para ello, Windows usa lo que se denomina un CSP, que es la capa software que implementa una interfaz conocida por el Sistema y que le permite a este realizar una serie de operaciones. Pues bien, si se pretendiera realizar un CSP propio para la interacción de Windows con la tarjeta, el código diseñado en este proyecto encargado de interactuar con PKCS#11 sería una fuente verdaderamente útil de obtener instrucciones de comunicación ya generadas y que podrían ser reutilizadas en el nuevo marco.

1.4 – OBJETIVOS DEL PROYECTO

Los objetivos para este proyecto son generar una aplicación que trate el estándar PKCS#11 enfocada a la infraestructura PKI de modo que se tenga un código fácil de modificar, expandir, portar y reutilizar. De forma más detallada los objetivos serían:

- **Comprensión del estándar PKCS#11**. Es imprescindible entender el funcionamiento de este estándar para así poder hacer un uso correcto de él y poder obtener los resultados deseados. Por tanto, el primer objetivo ha de ser la realización de un estudio de las capacidades que ofrece el estándar enfocadas a PKI (certificados, criptografía asimétrica, firmas digitales, etc.). Además también es necesario comprender la dinámica de su uso, de modo que se puedan entender el significado de las invocaciones y su correcto orden.

- **Obtención de códigos portables**. Aunque en principio el proyecto siempre se va a enmarcar dentro del sistema operativo Windows, se requiere que el código sea lo más portable a otros sistemas como Linux, ya que el uso de las tarjetas, en principio, debe ser totalmente independiente del sistema operativo. Además, se debe realizar un código usando, de forma flexible, el estándar PKCS#11 con el fin de garantizar una compatibilidad amplia con los distintos tipos de tarjetas y así evitar tener que incluir códigos concretos para el control de tarjetas específicas.

- **Gestión de los objetos de la tarjeta**. El usuario tiene que poder interactuar sobre las tarjetas y los objetos que contienen para, así, poder realizar las operaciones que sean requeridas con los elementos adecuados. Para ello, en principio, el usuario deberá poder crear, borrar, visualizar y modificar los objetos que estén en la tarjeta, teniendo siempre en cuenta las limitaciones del estándar PKCS#11.

- **Importación PKCS#12**. La aplicación tiene que ser capaz de importar perfiles PKCS#12, típicamente integrados en ficheros con extensión .pfx. Estos ficheros contienen certificados y el par de claves asociados al mismo (incluida la clave privada) y se utilizan para la transmisión de identidades digitales. Debido a que contienen claves privadas, estos ficheros típicamente se transmiten encriptados por medio de algún mecanismo criptográfico simétrico (como triple DES o AES). La aplicación tiene que ser capaz de incorporar claves y certificados extraídos de perfiles PKCS#12 a fin de almacenar dichas identidades en la tarjeta.

- **Generación de pares de claves**. Debe permitirse la generación de claves (pública y privada) dentro de la tarjeta, que podrán ser utilizadas para firma, verificación y generación de peticiones de certificados X509 (lo que se denominan CSR). Estas claves en la práctica siempre serán RSA, ya que las tarjetas por lo general no permiten otro tipo de claves y su longitud será normalmente de 1024 o 2048 bits, en función de las características hardware de la tarjeta en uso en cuestión.

- **Generación de CSR**. Una capacidad importante que debe ser explotada es la de poder generar CSR a partir de las claves. Ésta es la forma más habitual y primer paso para exportar e intercambiar claves públicas (parte fundamental en las arquitecturas PKI). Se tendrán que generar peticiones de dos tipos diferentes. Por un lado, se generaran peticiones generales, enfocadas a crear certificados de un uso estándar de firma digital. Por otro lado, se podrán generar peticiones enfocadas a la obtención de certificados de logon en sistemas Windows. Estos certificados de logon tienen la finalidad de poder acceder a un Windows sin necesidad de usuario y contraseña.

- **Firma y verificación de ficheros**. Ésta es, sin duda, la característica más importante de la aplicación, la capacidad de firmar ficheros (o buffers de datos) mediante las claves (privada para firmar y pública para verificar). La aplicación permitirá explotar los distintos mecanismos soportados en cada tarjeta, a fin de poder realizar firmas digitales y permitir su comprobación. Aunque el estándar PKCS#11 soporta varios paradigmas, en la práctica todas las SmartCards sólo soportan PKI basada en RSA.

1.5 – ESTRUCTURA DE LA MEMORIA

La memoria estará dividida en varios bloques que se describirán brevemente a continuación:

En una primera parte (la actual), se hace una introducción al proyecto, donde se expone el problema existente, cómo se va a afrontar su solución y se describen brevemente las tecnologías y mecanismos que se emplearán para la misma.

En una segunda parte, denominada como "Estado de arte", se realiza una descripción de todas las tecnologías, paradigmas, sistemas y algoritmos que son necesarios conocer para entender la memoria presente y el proyecto en general. Ahí se harán descripciones de las tarjetas usadas durante el proyecto, sus características, se hará un definición en profundidad de los estándares PKCS#11, 12 y 15, se hará una introducción para comprender qué son y cómo se manejan los CSR y certificados y se analizarán aplicaciones existentes para la gestión de tarjetas (como es el caso del proyecto OpenSC).

En el siguiente bloque de la memoria, se tratará el análisis de los diferentes problemas a abordar para la consecución de la aplicación de gestión del estándar PKCS#11 y los otros elementos que trata el presente proyecto. Aquí se hará uso de UML y estudios de requisitos para poder tener una buena visión de las necesidades. Tras el análisis, se realizará una descripción del diseño, donde se presentarán las soluciones tal como han sido concebidas, usando nuevamente UML para la descripción de los módulos, algoritmos, etc.

Tras el proceso de desarrollo, se realizará un estudio de las pruebas realizadas, los requisitos necesarios para considerarse como superadas y los resultados obtenidos. Se harán pruebas unitarias para comprobar el correcto funcionamiento de cada una de las funciones implementadas y pruebas de aceptación, donde se analizará si las funcionalidades de alto nivel funcionan adecuadamente.

Al final de todo esto, se realizará un análisis del proyecto desde una perspectiva de tiempos y recursos. Esto se realizará haciendo un estudio de cada una de las fases en las que se ha dividido el proyecto. Para esto, se hará uso de diagramas GANTT, donde se puede ver de forma gráfica y directa un resumen de cómo se ha ido elaborando el proyecto desde una perspectiva temporal y de recursos.

Tras esto, la última parte de la memoria estará dedicada a analizar las conclusiones del proyecto, futuros desarrollos que podrían realizarse a partir de todos los resultados obtenidos de este proyecto y unos apéndices donde se expondrán configuraciones, instalaciones y comandos necesarios para poder realizar pruebas, configurar entornos o similares necesarios para englobar la aplicación en los entornos

de ejecución. Aquí, se harán descripciones de las configuraciones y comandos de OpenSC (para la gestión de PKCS#11), se describirá cómo gestionar arquitecturas PKI con OpenSSL, cómo almacenar los certificados dentro de la arquitectura PKI de Windows y cómo debe configurarse un Windows (un 2003 Server) para poder realizar logon en él por medio de SmartCards. Los apéndices concluirán con el código fuente que compone la aplicación.

2 – TECNOLOGÍAS SOFTWARE

En esta sección de tecnologías software, se va a hacer una descripción de todos los proyectos y aplicaciones de los que se ha hecho uso para la realización de este proyecto, así como también de todos los estándares, algoritmos, codificaciones, etc., que deben ser tenidos en cuenta para poder entender este proyecto. Los puntos básicos a tratar en este apartado van a ser:

- El estándar de generación de peticiones de certificados (PKCS#10). Donde se tratará cómo se codifican, qué información contienen y el proceso que debe cumplirse para que una petición se convierta en un certificado.
- La arquitectura PKI (infraestructura de clave pública), donde se hablará de cómo funciona la arquitectura de clave pública, qué elementos la componen, cómo se gestiona la confianza y los elementos necesarios para crear una PKI (tales como una CA, certificados, claves, etc.).
- RSA. En este apartado, se tratará en qué consiste el algoritmo RSA y sus principios, tanto a nivel de cifrado como de firma digital.
- PKCS#11. Este será el punto principal de las tecnologías software, donde se expondrá en qué consiste realmente este estándar de gestión de tokens criptográficos y las posibilidades que ofrece.
- PKCS#12. En él se describirá en qué consiste este estándar de intercambio de perfiles, qué información se transfiere, cómo se asegura, etc.
- PKCS#15. Este será un apartado informativo, ya que realmente no se trata en el desarrollo del proyecto. El estándar en sí trata sobre cómo se almacena la información dentro de una SmartCard. PKCS#15 se podría comparar a cómo es el sistema de ficheros de la tarjeta.
- OpenSSL. Este es un proyecto abierto que permite la gestión de arquitecturas PKI en toda su extensión, desde las claves hasta las CAs. Este proyecto (o mejor dicho su API) se usará para la gestión de los certificados que se hace dentro de las tarjetas.

2.1 – PKCS#10 Y CERTIFICADOS

PKCS#10 es un estándar de los laboratorios RSA en el que se define cómo deben generarse las peticiones de certificados.

La información de un PKCS#10 sigue una estructura ASN.1 (que se verá después) donde básicamente hay una serie de datos que estarán asociados al futuro certificado. Toda petición ha de tener:

- Información relativa al solicitante del certificado (como puede ser el nombre, el país, el correo, etc.).
- Detalles sobre la clave pública, tales como:
  - Algoritmos de la clave pública.
  - Módulo de la clave pública.
  - Exponente de la clave pública.
- Usos de la clave, en otras palabras, que podrá comprobarse con el futuro certificado (firmas digitales, firmas de certificados, logon en sistemas operativos, etc.
- Finalmente toda petición de certificado ha de llevar un campo con la firma del solicitante del certificado. Esta firma se realizará con la clave privada asociada a la clave pública que contiene el certificado, de modo que cualquier CA que pretenda firmar el certificado pueda comprobar que el solicitante del certificado posee la clave privada asociada al mismo (se garantiza el no repudio) y además se verifica que la petición de certificado no se ha modificado o estropeado hasta llegar a la CA (se garantiza la integridad de la misma).

Una petición de certificado no tiene validez en ningún ámbito PKI, ya que, para ello, debe ser aceptada por una CA. En otras palabras, en una petición el usuario incluye toda la información que espera que contenga su futuro certificado y esta información es enviada a una CA. Cuando la CA la acepta y la firma, entonces se obtendrá un certificado válido con toda la información que el usuario generó; pero, ahora sí, siendo válida para ser usada en arquitecturas PKI.

2.1.1 – ASN.1

ASN.1 (Abstract Sintaxis Notation o Notación de Sintaxis Abstracta) Es una norma de representación de datos independiente de la plataforma. ASN.1 tiene por tanto una forma de representar todos los tipos de datos básicos como enteros o cadenas de caracteres de forma que sean entendidos del mismo modo en máquinas con arquitectura hardware diferente.

ASN.1 sigue una estructura de árbol para ordenar la información en su interior y para realizar la codificación sigue el formato BER. BER sólo entra en cómo debe ser ordenada la información (es la estructura de ASN.1), pero no define, en sí, cómo debe ser codificada físicamente dicha información (eso se verá más adelante). ASN.1 realmente no usa las normas BER en toda su extensión, sino que realmente usa un subgrupo de las mismas que se conoce como DER.

La estructura las secuencias DER es recursiva, de modo que el propio documento en un nodo en sí y a su vez se va subdividiendo en nuevos nodos que siguen el mismo patrón que el nodo inicial. La estructura de dichos nodos es:

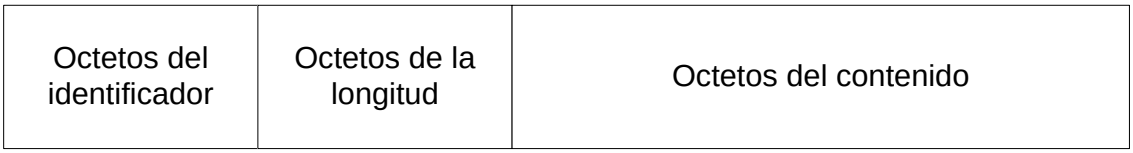

Ilustración 1: Estructura DER

El identificador define qué tipo de nodo estamos tratando. Básicamente hay dos tipos de nodos a representar: los tipos construidos y los primitivos. Los tipos construidos no contienen información por sí mismos, son siempre una secuencia que contiene en su interior otros nodos que pueden ser nuevos tipos construidos o tipos primitivos. Por el contrario, los tipos primitivos sí que contienen información, ya sean enteros, fechas, cadenas, etc. Todos los tipos se representan por una secuencia de 8 bits que los identifica. Los tipos principales usados en la codificación de CSR y certificados X.509 son:

Nombre de la clase	Tipo	Valor Hexadecimal	Descripción
SEQUENCE	Construido	30	Es una lista ordenada de tipos.
OBJECT IDENTIFIER	Primitivo	06	Representa los indicadores de los objetos dentro del árbol (tales como algoritmos, uso de certificados, etc.).
SET	Construido	31	Es una lista no ordenada de tipos, donde todos han de ser diferentes.
IA5	Primitivo	16	Cadena de caracteres de 7 bits, codificada dentro de 8 bits.
PRINTABLE STRING	Primitivo	13	Cadenas de caracteres Imprimibles.
UTC TIME	Primitivo	17	Representación de tiempo.
NULL	Primitivo	05	Tipo nulo, usado generalmente como

Nombre de la clase	Tipo	Valor Hexadecimal	Descripción
			delimitador.
BIT STRNG	Construido / Primitivo	03	Cadena que no es tratada a nivel de bytes sino de bits.
OCTECT STRING	Primitivo	04	Cadenas de caracteres de 8 bits.
BOOLEAN	Primitivo	01	Representa valores falsos (el 0) y verdaderos (cualquier otro valor).
INTEGER	Primitivo	02	Representa números enteros.
UTF8 STRING	Primitivo	0C	Es un String codificado con el formato UTF de 8 bits.

Tabla 1: Tipos DER básicos

La longitud de los elementos DER se expresa normalmente con 1 byte, quedando sin uso aquel que tiene todo el octeto compuesto por 1s. Esto presenta el inconveniente de que sólo permite representar longitudes de 1 a 127, pero por la composición del árbol ASN.1 esto se queda muy corto. Para representar longitudes mayores, se usan más bytes siguiendo la siguiente notación:

El primer byte deja su bit más significativo a 1 (por eso en las longitudes normales no se pueden usar las cadenas de todo 1s). El resto de bits representan el número de bytes consecutivos que hay que tener en cuenta para calcular la longitud. A modo de ejemplo una longitud de 35 se representaría:

$35_{10} = 00100011_2$

La longitud 130 (mayor que 127) se representaría como:

10000001 (representa una posición para la longitud)

$130_{10} = 10000010_2$

Por tanto la longitud final sería:

10000001 10000010

Poniéndolo todo junto, y a modo de ejemplo, así se expresaría la secuencia de un certificado que contiene la información correspondiente a la clave pública. Para reducir espacio se usará una notación hexadecimal en lugar de la binaria.

***30** 81 89 **02** 81 81 00 BF B5 69 D3 6D 84 36 E8 8F*
4B A1 50 83 4D 96 51 C4 95 FF 54 70 DD BB 12 EB
74 0F FB DC 6E DB 43 42 AB 06 A1 72 A7 EE 5F 1C
B2 E2 2A CD 75 1C 5D 3A 79 50 67 EE B3 07 98 F4
CA 85 28 1A 14 87 03 3B B8 DC 9F 7C A1 B3 3F DD
D1 BD D1 01 E0 7A 19 A8 9F 55 FC 2D A5 39 49 5E

B5 99 D0 9C EE CB 8E EC CA DD B2 0B 87 1C 90 5E
02 24 CC D8 87 06 81 9A 38 91 71 0B 42 14 2E 71
*E5 60 07 B1 19 8C EF **02** 03 01 00 01*

El primer 30 representa que tenemos una secuencia. El 81 89 siguientes que tenemos una longitud, el 81 nos dice que la longitud tiene 1 byte, que es el 89_{16} que en decimal es 137. Luego tenemos un 02, que representa que tenemos un dato primitivo, un entero, su longitud es 81 81 que representa el 129 en decimal. El 00 siguiente es un convenio de escritura del módulo de las claves. Los 128 números hexadecimales siguientes se corresponden con el módulo de la clave privada. El 02 siguiente (pasados esos 128 números) representa que tenemos nuevamente un entero, el exponente de la clave. El 03 nos dice la longitud es directamente 3 bytes y los 3 hexadecimales siguientes (01 00 01) son el exponente (concretamente 65537 en decimal).

La forma de expresar las secuencias dentro de ASN.1 es siguiendo una BNF de una gramática. Una representación ASN.1 de un certificado X.509 sería:

```
Certificate ::= SIGNED SEQUENCE{
           version [0]     Version DEFAULT v1988,
           serialNumber    CertificateSerialNumber,
           signature       AlgorithmIdentifier,
           issuer          Name,
           validity        Validity,
           subject         Name,
           subjectPublicKeyInfo    SubjectPublicKeyInfo}


   Version ::=     INTEGER {v1988(0)}

   CertificateSerialNumber ::=     INTEGER


   Validity ::=    SEQUENCE{
           notBefore       UTCTime,
           notAfter        UTCTime}

   SubjectPublicKeyInfo ::=        SEQUENCE{
           algorithm               AlgorithmIdentifier,
           subjectPublicKey        BIT STRING}

   AlgorithmIdentifier ::= SEQUENCE{
           algorithm       OBJECT IDENTIFIER,
           parameters      ANY DEFINED BY algorithm OPTIONAL}
```

Como se observa en la gramática, los Certificados X.509 se componen en primer lugar de un entero con la versión (que es generalmente la 3), luego un Número de Serie que es el identificador que le asignó la CA cuando firmó el certificado. A continuación, una secuencia con los usos de la clave, los datos del emisor (la CA en

cuestión que firmó el certificado), el periodo de validez del certificado (la fecha de inicio de validez y la fecha de expiración), luego vienen los datos del propietario del certificado y, a continuación, iría la clave pública. Estos datos no tienen por qué seguir ese orden, pero generalmente estarán todos (aunque no todos son imprescindibles). El último elemento de un certificado es la firma de todo el buffer de datos con la clave privada de la CA.

El caso de las CSR sería muy similar, pero sin la información asociada al certificado en sí, como podría ser el Número de Serie o la firma de la CA (tiene la firma del solicitante en su lugar).

Un certificado completo ASN.1 típico sería similar al que sigue:

```
Offset| Len  |LenByte|
======+======+=======+=================================================================
     0|   786|      3| SEQUENCE :
     4|   635|      3|    SEQUENCE :
     8|     3|      1|       CONTEXT SPECIFIC (0) :
    10|     1|      1|          INTEGER : 2
    13|     1|      1|       INTEGER : 5
    16|    13|      1|       SEQUENCE :
    18|     9|      1|          OBJECT IDENTIFIER : sha1withRSAEncryption [1.2.840.113549.1.1.5]
    29|     0|      1|          NULL : ''
    31|   139|      2|       SEQUENCE :
    34|    11|      1|          SET :
    36|     9|      1|             SEQUENCE :
    38|     3|      1|                OBJECT IDENTIFIER : countryName [2.5.4.6]
    43|     2|      1|                PRINTABLE STRING : 'ES'
    47|    15|      1|          SET :
    49|    13|      1|             SEQUENCE :
    51|     3|      1|                OBJECT IDENTIFIER : stateOrProvinceName [2.5.4.8]
    56|     6|      1|                PRINTABLE STRING :
      |      |       |                   'Madrid'
    64|    16|      1|          SET :
    66|    14|      1|             SEQUENCE :
    68|     3|      1|                OBJECT IDENTIFIER : localityName [2.5.4.7]
    73|     7|      1|                PRINTABLE STRING :
      |      |       |                   'Leganes'
    82|    13|      1|          SET :
    84|    11|      1|             SEQUENCE :
    86|     3|      1|                OBJECT IDENTIFIER : organizationName [2.5.4.10]
    91|     4|      1|                PRINTABLE STRING :
      |      |       |                   'uc3m'
    97|    16|      1|          SET :
    99|    14|      1|             SEQUENCE :
   101|     3|      1|                OBJECT IDENTIFIER : organizationalUnitName [2.5.4.11]
   106|     7|      1|                PRINTABLE STRING :
      |      |       |                   'evalues'
   115|    20|      1|          SET :
   117|    18|      1|             SEQUENCE :
   119|     3|      1|                OBJECT IDENTIFIER : commonName [2.5.4.3]
   124|    11|      1|                PRINTABLE STRING :
      |      |       |                   'J.M. Sierra'
   137|    34|      1|          SET :
   139|    32|      1|             SEQUENCE :
   141|     9|      1|                OBJECT IDENTIFIER : emailAddress [1.2.840.113549.1.9.1]
   152|    19|      1|                IA5 STRING :
      |      |       |                   'jm.sierra@gmail.com'
   173|    30|      1|       SEQUENCE :
   175|    13|      1|          UTC TIME : '0911302320592'
   190|    13|      1|          UTC TIME : '1011302320592'
   205|   133|      2|       SEQUENCE :
   208|    11|      1|          SET :
   210|     9|      1|             SEQUENCE :
   212|     3|      1|                OBJECT IDENTIFIER : countryName [2.5.4.6]
   217|     2|      1|                PRINTABLE STRING : 'ES'
   221|    15|      1|          SET :
   223|    13|      1|             SEQUENCE :
   225|     3|      1|                OBJECT IDENTIFIER : stateOrProvinceName [2.5.4.8]
   230|     6|      1|                PRINTABLE STRING :
      |      |       |                   'Madrid'
   238|    16|      1|          SET :
   240|    14|      1|             SEQUENCE :
   242|     3|      1|                OBJECT IDENTIFIER : localityName [2.5.4.7]
   247|     7|      1|                PRINTABLE STRING :
      |      |       |                   'Leganes'
   256|    13|      1|          SET :
   258|    11|      1|             SEQUENCE :
   260|     3|      1|                OBJECT IDENTIFIER : organizationName [2.5.4.10]
   265|     4|      1|                PRINTABLE STRING :
      |      |       |                   'uc3m'
   271|    16|      1|          SET :
```

```
  273|    14|      1|            SEQUENCE :
  275|     3|      1|               OBJECT IDENTIFIER : organizationalUnitName [2.5.4.11]
  280|     7|      1|               PRINTABLE STRING :
     |      |       |                  'evalues'
  289|    14|      1|         SET :
  291|    12|      1|            SEQUENCE :
  293|     3|      1|               OBJECT IDENTIFIER : commonName [2.5.4.3]
  298|     5|      1|               PRINTABLE STRING :
     |      |       |                  'Nacho'
  305|    34|      1|         SET :
  307|    32|      1|            SEQUENCE :
  309|     9|      1|               OBJECT IDENTIFIER : emailAddress [1.2.840.113549.1.9.1]
  320|    19|      1|               IA5 STRING :
     |      |       |                  'ignacioas@gmail.com'
  341|   159|      2|      SEQUENCE :
  344|    13|      1|         SEQUENCE :
  346|     9|      1|            OBJECT IDENTIFIER : rsaEncryption [1.2.840.113549.1.1.1]
  357|     0|      1|            NULL : ''
  359|   141|      2|         BIT STRING UnusedBits:0 :
  363|   137|      2|            SEQUENCE :
  366|   129|      2|               INTEGER :
     |      |       |                  00BFB569D36D8436E88F4BA150834D9651C495FF54
     |      |       |                  70DDBB12EB740FFBDC6EDB4342AB06A172A7EE5F1C
     |      |       |                  B2E22ACD751C5D3A795067EEB30798F4CA85281A14
     |      |       |                  87033BB8DC9F7CA1B33FDDD1BDD101E07A19A89F55
     |      |       |                  FC2DA539495EB599D09CEECB8EECCADDB20B871C90
     |      |       |                  5E0224CCD88706819A3891710B42142E71E56007B1
     |      |       |                  198CEF
  498|     3|      1|               INTEGER : 65537
  503|   137|      2|      CONTEXT SPECIFIC (3) :
  506|   134|      2|         SEQUENCE :
  509|     9|      1|            SEQUENCE :
  511|     3|      1|               OBJECT IDENTIFIER : basicConstraints [2.5.29.19]
  516|     2|      1|               OCTET STRING :
  518|     0|      1|                  SEQUENCE : ''
  520|    44|      1|            SEQUENCE :
  522|     9|      1|               OBJECT IDENTIFIER : netscape-comment [2.16.840.1.113730.1.13]
  533|    31|      1|               OCTET STRING :
  535|    29|      1|                  IA5 STRING :
     |      |       |                     'OpenSSL Generated Certificate'
  566|    11|      1|            SEQUENCE :
  568|     3|      1|               OBJECT IDENTIFIER : keyUsage [2.5.29.15]
  573|     4|      1|               OCTET STRING :
  575|     2|      1|                  BIT STRING UnusedBits:7 :
     |      |       |                     80
  579|    29|      1|            SEQUENCE :
  581|     3|      1|               OBJECT IDENTIFIER : subjectKeyIdentifier [2.5.29.14]
  586|    22|      1|               OCTET STRING :
  588|    20|      1|                  OCTET STRING :
     |      |       |                     C142822D0053C94C72AC3E766890BB7FB3DEF8A
     |      |       |                     1
  610|    31|      1|            SEQUENCE :
  612|     3|      1|               OBJECT IDENTIFIER : authorityKeyIdentifier [2.5.29.35]
  617|    24|      1|               OCTET STRING :
  619|    22|      1|                  SEQUENCE :
  621|    20|      1|                     CONTEXT SPECIFIC (0) :
     |      |       |                        'Fä•ëÒë¶7ûšÍå/7µ_Ä,þö'
  643|    13|      1|   SEQUENCE :
  645|     9|      1|      OBJECT IDENTIFIER : sha1withRSAEncryption [1.2.840.113549.1.1.5]
  656|     0|      1|      NULL : ''
  658|   129|      2|   BIT STRING UnusedBits:0 :
     |      |       |      87634F5B912FDE6291F93E82D1BDECA75EC7965D30EE6384637C67
     |      |       |      F7965600D2A018A7801A03F7DE66DA2E3047EAFC217A028C16E8C1
     |      |       |      F237A681F80DE6AD04357FA7142A7FC9A2018D19A2DDBD54E2CBA7
     |      |       |      3F6A980BFFE238B5A5EA9D6C18533AD91C261A6B7CE6DEC5770904
     |      |       |      B74D90604B059499397B4950E1C20CB54F2D1609
```

A grandes rasgos, el certificado se compone de una Sequence que contiene dos Sequences más. La última, contiene la firma del certificado por parte de la CA y la primera que se divide en varios grupos de Sequences: una para los datos del propietario del certificado, otra para el periodo de validez del certificado, un grupo de Set con los usos del certificado y, posteriormente, otra Sequence con la clave pública.

2.1.2 – CODIFICACIÓN DE CERTIFICADOS Y CSR

Todas las secuencias y tipos que se han visto en el apartados anterior, con sus respectivos códigos asociados tienen varias formas de codificarse físicamente (dentro de ficheros) para ser entendidas y tratadas por las respectivas aplicaciones que usen

tanto las peticiones PKCS#10, como los propios certificados. Estas formas básicamente son dos, DER Encoding y PEM o Base 64.

DER Encoding básicamente consiste en una representación binaria directa de la codificación DER vista anteriormente. De esta forma cada byte de cada elemento de la codificación DER es almacenado directamente en su equivalente ASCII. De este modo, si hubiera que codificar el número 30_{16} (correspondiente al código de Secuencia) sería el ASCII 0 (48 decimal). El resultado es un buffer de bytes difícilmente identificable por una persona.

La codificación PEM no es que sea inteligible de por sí, pero en el caso de elementos PKI (como certificados o CSR) se le añaden unas cabeceras a los bytes codificados de modo que, aunque no se entienda el contenido, se pueda ver qué es lo que está codificado (un certificado por ejemplo). Esta codificación PEM no escribe los bytes traducidos a ASCII, sino su codificación en Base 64 (la misma utilizada frecuentemente en correos MIME). Esta codificación funciona del siguiente modo:

- Se agrupan los datos en grupos de 24 bits (lo que serían 3 caracteres ASCII).
- Se divide en grupos de 6 bits y a cada secuencia de esos 6 bits se le asigna un carácter de una tabla de conversión. Se deben añadir 0s para rellenar y hacer que la traducción se haga en bloques de 6 bits.
- Se añade un padding con el carácter ASCII = (no contemplado en la tabal de traducción) para garantizar que Base 64 codifica en bloques de 4 caracteres (24 bits de entrada, en grupos de 6, da como resultado 4 caracteres).

PEM añade una cabecera de inicio, luego incluye el buffer en codificación Base 64 y, finalmente, una cabecera de final.

VALOR	CARACTER	VALOR	CARACTER	VALOR	CARACTER	VALOR	CARACTER
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+

VALOR	CARACTER	VALOR	CARACTER	VALOR	CARACTER	VALOR	CARACTER
15	P	31	f	47	v	63	/

Tabla 2: Codificación Base 64

Las cabeceras PEM para las request son:

```
-----BEGIN CERTIFICATE REQUEST-----
-----END CERTIFICATE REQUEST-----
```

Las cabeceras PEM para los certificados X.509 son:

```
-----BEGIN CERTIFICATE-----
-----END CERTIFICATE-----
```

Un certificado codificado en formato PEM, tendría el siguiente aspecto:

```
-----BEGIN CERTIFICATE-----
MIIDEjCCAnugAwIBAgIBBTANBgkqhkiG9w0BAQUFADCBizELMAkGA1UEBhMCRVMx
DzANBgNVBAgTBk1hZHJpZDEQMA4GA1UEBxMHTGVnYW5lczENMAsGA1UEChMEdWMz
bTEQMA4GA1UECxMHZXZhbHVlczEUMBIGA1UEAxMLSi5NLiBTaWVycmExIjAgBgkq
hkiG9w0BCQEWE2ptLnNpZXJyYUBnbWFpbC5jb20wHhcNMDkxMTMwMjMyMDU5WhcN
MTAxMTMwMjMyMDU5WjCBhTELMAkGA1UEBhMCRVMxDzANBgNVBAgTBk1hZHJpZDEQ
MA4GA1UEBxMHTGVnYW5lczENMAsGA1UEChMEdWMzbTEQMA4GA1UECxMHZXZhbHVl
czEOMAwGA1UEAxMFTmFjaG8xIjAgBgkqhkiG9w0BCQEWE2lnbmFjaW9hc0BnbWFp
bC5jb20wgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAL+1adNthDboj0uhUINN
llHElf9UcN27Eut0D/vcbttDQqsGoXKn7l8csuIqzXUcXTp5UGfusweY9MqFKBoU
hwM7uNyffKGzP93RvdEB4HoZqJ9V/C2lOUletZnQnO7LjuzK3bILhxyQXgIkzNiH
BoGaOJFxC0IULnHlYAexGYzvAgMBAAGjgYkwgYYwCQYDVR0TBAIwADAsBglghkgB
hvhCAQ0EHxYdT3BlblNTTCBHZW5lcmF0ZWQgQ2VydGlmaWNhdGUwCwYDVR0PBAQD
AgeAMB0GA1UdDgQWBBTBQoItAFPJTHKsPnZokLt/s974oTAfBgNVHSMEGDAWgBRG
5JXr0uu2N/uazeUvN7VfxCz+9jANBgkqhkiG9w0BAQUFAAOBgQCHY09bkS/eYpH5
PoLRveynXseWXTDuY4RjfGf3llYA0qAYp4AaA/feZtouMEfq/CF6AowW6MHyN6aB
+A3mrQQ1f6cUKn/JogGNGaLdvVTiy6c/apgL/+I4taXqnWwYUzrZHCYaa3zm3sV3
CQS3TZBgSwWUmTl7SVDhwgy1Ty0WCQ==
-----END CERTIFICATE-----
```

2.1.3 – DETALLES DE LOS CERTIFICADOS

En este proyecto, tanto las peticiones PKCS#10 que se generen, como los certificados (ya firmados por una CA) que se usen en la SmartCard, deben estar codificados en formato PEM. Además, todos los datos extraídos de la SmartCard a fichero, siempre tendrán codificación PEM.

Se va a trabajar con dos tipos diferentes de certificados: los certificados genéricos, pensados para el uso de firma digital y los certificados de logon, pensados para la autenticación en sistemas Windows.

Los certificados genéricos tendrán las siguientes características:

1. Como datos del solicitante incluidos en el Distinguished Name serán:
    a. "Country Name" o País
    b. "State or Province Name" o Provincia
    c. "Locality Name" o Ciudad
    d. "Organization Name" u Organización
    e. "Organizational Unit Name" o Unidad
    f. "Common Name" o Nombre
    g. "Email Address" o Dirección de correo
2. El formato de las claves será RSA de 1024 ó 2048 bits.
3. El algoritmo de firma será SHA1 con RSA.
4. Los usos de la clave serán:
    a. "Digital Signature" o Firma Digital
    b. "Non Repudiation" o No Repudio
    c. "Key Encipherment" o Cifrado de Clave

Los certificados de logon por el contrario contendrán la siguiente información (que es toda la información necesaria para que un certificado sea válido para hacer login en un sistema Windows):

1. Como datos del solicitante incluidos en el Distinguished Name serán:
    a. Common Name con el nombre del usuario
    b. Common Name con el nombre del grupo de usuarios
    c. DC con el nombre de la máquina
    d. DC con la extensión de la máquina
    e. UPN con el formato usuario@Dominio.extensión

2. El formato de las claves será RSA de 1024 ó 2048 bits.
3. El algoritmo de firma será SHA1 con RSA.
4. Los usos de la clave serán:
    a. “Digital Signature” o Firma Digital
    b. “Non Repudiation” o No Repudio
    c. “Key Encipherment” o Cifrado de Clave
5. Los usos mejorados de la clave serán:
    a. Autenticación de cliente
    b. Inicio de sesión de tarjeta inteligente

2.2 – PKI

La infraestructura de clave pública consiste en una serie de elementos hardware, software y procedimientos para ofrecer una serie de servicios basados en la criptografía asimétrica.

La gran ventaja de la criptografía asimétricas, es que el usuario, entidad o quien sea, posee una clave privada propia que nadie conoce y una clave pública complementaria a esa clave privada que es conocida en principio por cualquiera. Esta clave privada es divulgada por el entorno haciendo uso de los certificados digitales (el más usado es el X.509).

La PKI se basa realmente en la gestión de la confianza. Todo el mundo confía en una serie de personas/entidades que, a su vez, confían en terceras de modo que se establecen una serie de relaciones de confianza que hacen posible que finalmente alguien pueda confiar en la validez de datos de un tercero con el que, a priori, no tenía trato.

El punto básico de las PKI son los certificados, ya que en ellos se publica la clave pública de los sujetos. Para que dos elementos puedan confiar uno en otro, han de ponerse de acuerdo, ese acuerdo es que ambos van a confiar en un tercero (al que se conoce como CA o Autoridad de Certificación). La CA lo que hace es aceptar los CSR (datos con claves públicas) y los firma. Esa firma es la clave de la confianza, porque cualquiera tiene acceso a la clave pública de la CA, de modo que puede comprobar si un certificado de un particular ha sido realmente firmado por esa CA. En caso de comprobar la firma del certificado del particular y ser correcta puede confiar en que el certificado es de quien dice ser.

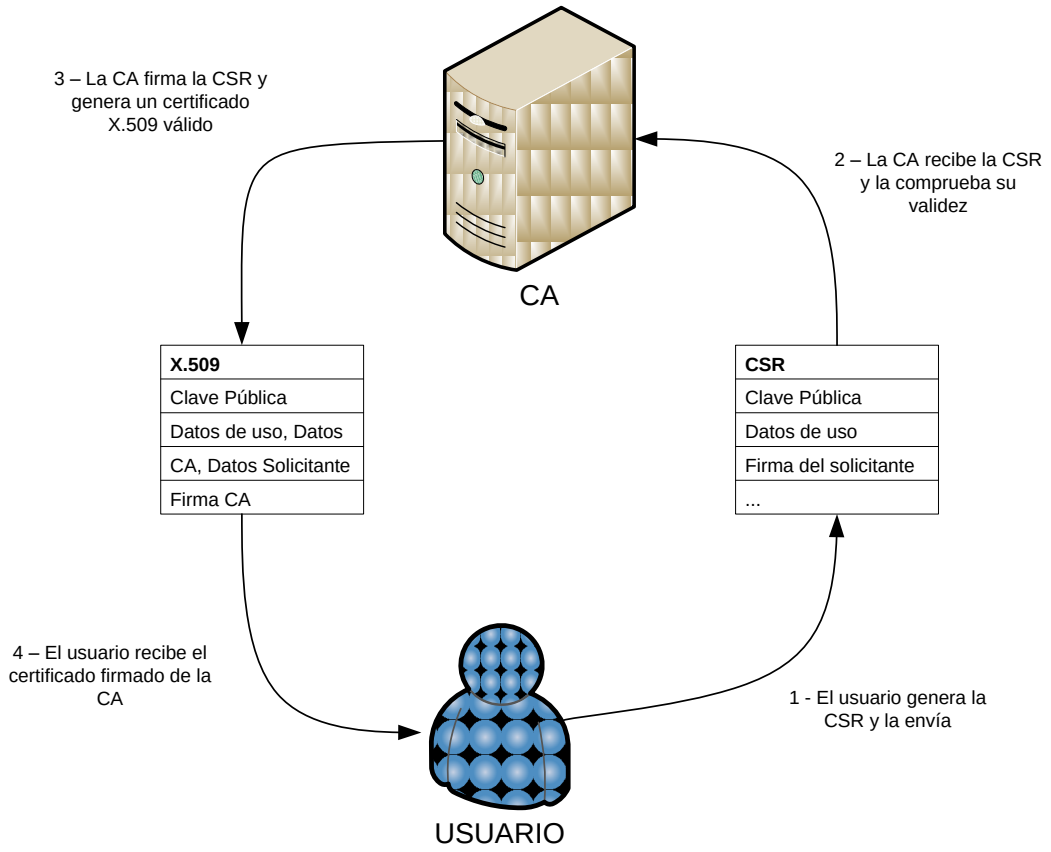

Ilustración 2: Proceso de obtención de un certificado X.509

El esquema antes visto es la base de PKI, pero sería tal cual inaplicable, ya que no todo el mundo podría confiar en una única CA porque haría de ella algo demasiado grande. Por tanto, aparecen CAs intermedias, de modo que se puede confiar en una CA que a su vez confíen en otras CAs, etc., de tal manera que lo que se tiene es una cadena de confianza que puede ser validada y comprobada.

Si la confianza se establece por medio de que alguien en que confiamos firma los contenedores de claves públicas de terceros, ¿dónde se establece el origen de esa confianza? Pues el origen se denomina CA raíz (y de hecho hay una infinidad de ellas). Una CA raíz es aquella que se autofirma el certificado y por tanto es el punto de partida de la cadena de seguridad.

Un entorno PKI nos permite en principio usar los siguientes servicios:

- Cifrado de datos, ya que yo puedo cifrar algo con la clave pública de alguien y sólo ese alguien podrá descifrar los datos (por medio de su clave privada). Por tanto, PKI puede ofrecer **confidencialidad**.
- Firma digital. En la que un usuario firma (cifra con su clave privada) un hash de datos que podrá ser validado por cualquiera que tenga su clave pública. De modo que se ofrece **integridad** de los datos (si los datos cambian la firma no

coincidiría) y **no repudio**, ya que sólo el propietario de la clave privada puede usarla, así que algo firmado no puede ser suplantado por terceros.

- Autenticación. Por medio del uso de certificados y claves privadas se puede permitir el acceso de personas a sistemas. Hay infinidad de posibles métodos de hacerse (mediante reto respuesta, comprobación de una firma y una clave privada, etc.). De modo que se ofrece el servicio de **autenticación**.

Para garantizar el adecuado funcionamiento de las PKI y evitar problemas de seguridad, los certificados tienen periodos de validez, fuera de los cuales nadie debería confiar en ellos, aunque si confíen en la CA que los firmó. Pasado el periodo de validez, los certificados deben ser renovados, caso en el que la CA firma la misma CSR pero con un nuevo periodo de validez, o cambiados directamente por otro certificado nuevo. El periodo de certificación estándar para un certificado suele ser de 1 año para los certificados de uso general y de 5 a 10 años para las CAs.

No sólo se asegura el buen funcionamiento de una PKI con esas prácticas, ya que las claves de usuarios han podido ser comprometidas (alguien tenga acceso a la clave privada de otra persona o alguna situación similar). Las CAs publican lo que se llama CRL (Certificate Revocation List o Listas de Revocación de Certificados). La revocación de un certificado consiste en que una CA dice, a todo el que la consulte, que un certificado concreto, aunque fuera firmado por ella, ya no tiene validez. Un certificado, una vez es revocado, queda publicado en la CRL de la CA de modo que, si alguien quiere consultar sobre dicho certificado, obtendrá que aparece en la CRL y, por tanto, que no es un certificado válido.

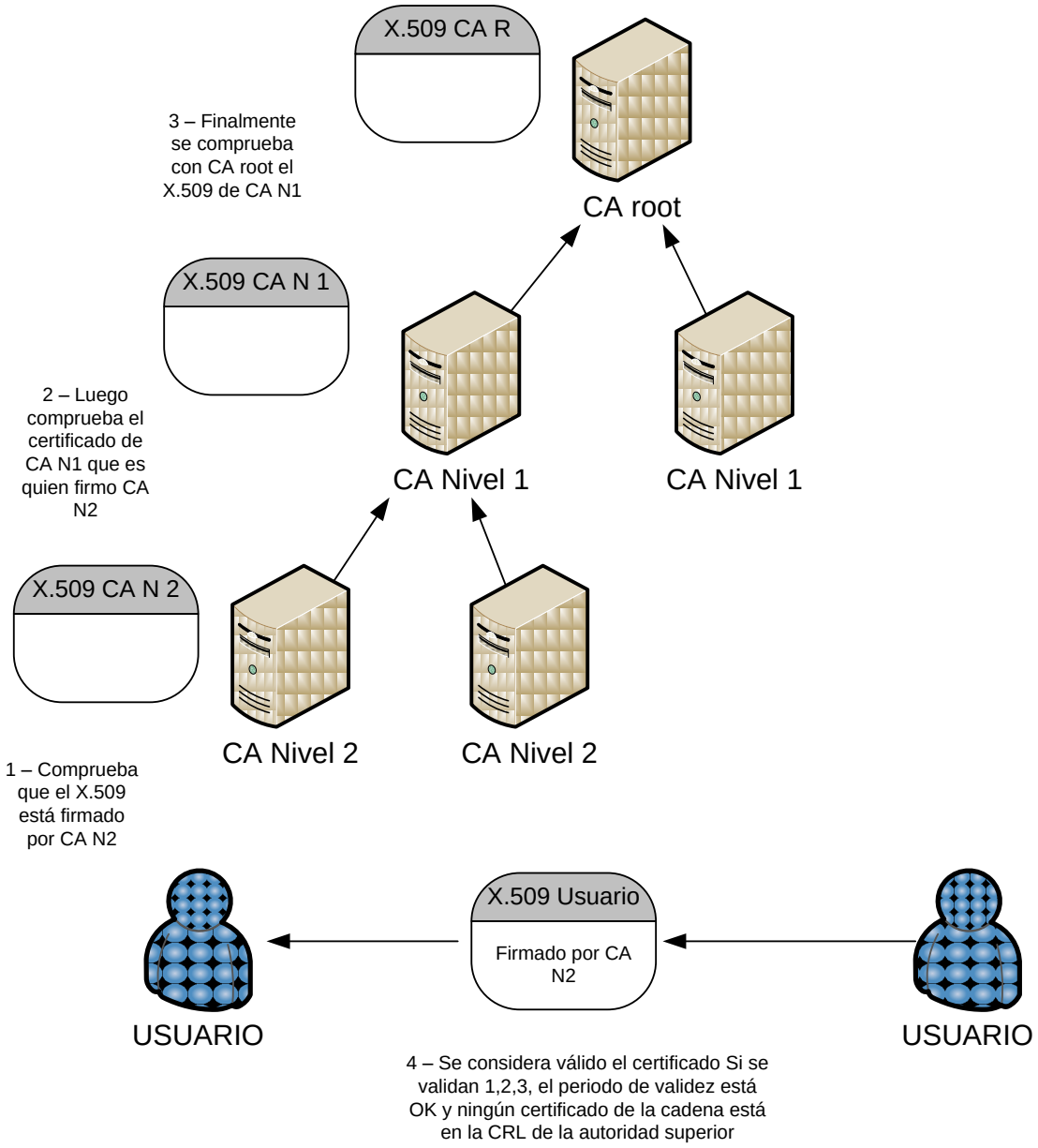

Ilustración 3: Cadenas de confianza PKI

En algunos entornos, como sucede en PKI gestionadas en Windows Server, existen las RA (Registration Authority o Autoridad de Registro) que son las encargadas de generar los CSR de los usuarios (no lo haría el propio usuario, sino que la RA lo haría por él) y también se encargaría de mediar con la CA para obtener un certificado firmado válido.

2.3 – RSA

RSA es un algoritmo creado por los laboratorios con el mismo nombre, basado en criptografía de clave pública (donde hay una clave pública y otra privada) que es utilizado tanto para cifrar como para firmar (especialmente para firmar).

RSA se basa en el concepto de función irreversible de modo que, recorrer el camino en el sentido contrario al que se aplicó, resulta imposible. En el caso de RSA su fortaleza radica en la factorización de números enteros obtenidos como múltiplos de primos enormes, de modo que a partir de una clave pública obtener su privada equivalente, sin tener "la trampa del criptosistema" (la trampa es una información utilizada en el proceso de generar las claves que o es destruida o no es accesible), es prácticamente imposible.

Los sistemas de clave pública requieren de claves enormes frente a criptografía simétrica, de modo que RSA debe usar claves de, como mínimo, 1024 bits para ser seguras (para operaciones más críticas como las realizadas por CA, se recomiendan claves de, al menos, 2048 bits).

En un escenario de cifrado (poco extendido) una persona utiliza la clave pública de alguien para cifrar un mensaje. Ese mensaje cifrado sólo podrá ser descifrado con la clave privada asociada a la pública con la que se cifró el mensaje. Este escenario ofrece confidencialidad

El caso más extendido es el de la firma digital. La firma digital consiste en que se cifra con la privada en lugar de con la pública. Esto supone que todo el mundo puede descifrar esos datos cifrados con la clave pública (que está distribuida y es accesible por todos) y es ahí donde esta justamente lo interesante, sólo el propietario de la clave privada puede hacer la firma (hay que tener la clave privada y sólo el propietario la tiene) y se puede comprobar dicha firma con la clave pública (que cualquiera puede hacerlo). La firma no se realiza sobre todo el mensaje (como se hace en el caso del cifrado), sino que se realiza sobre un resumen del mismo. Este resumen se realiza generalmente con las funciones hash MD5 o SHA1. La firma digital ofrece integridad (ofrecida por las funciones hash) y no repudio (obtenido al ser necesario el uso de la clave privada para la firma).

La siguiente ilustración describe el proceso de firma de un documento y su posterior verificación:

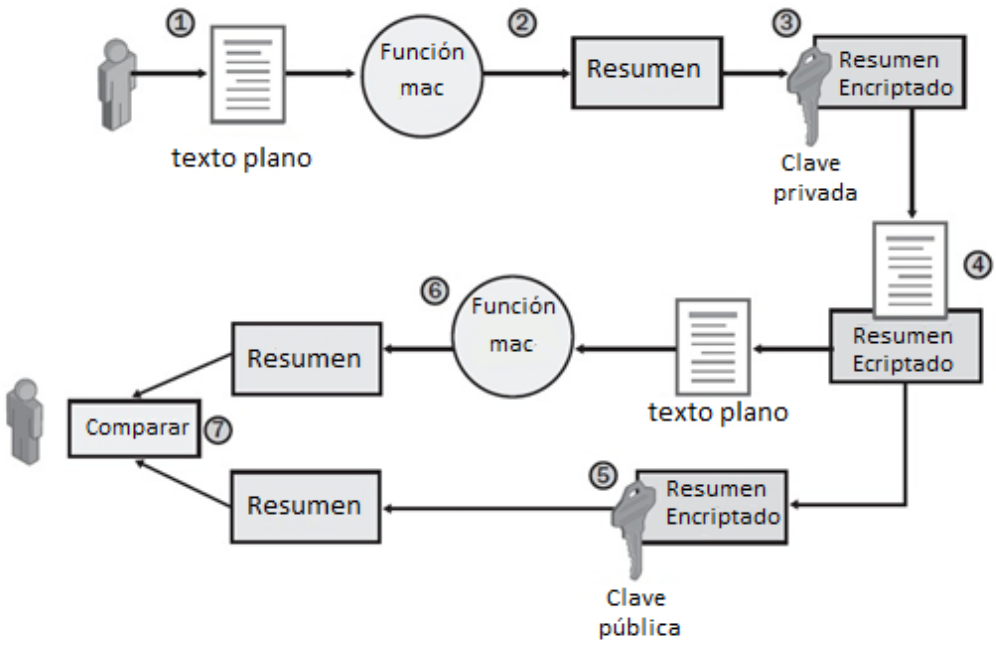

Ilustración 4: Firmado y comprobación

1. El usuario tiene el fichero, correo, texto, etc. que desea firmar.
2. Aplica una función resumen sobre el documento (MD5, SHA1, etc.).
3. El usuario cifra, con su clave privada, ese resumen. Esto es la firma del documento.
4. El usuario envía el documento original con la firma.
5. El destinatario recoge la firma y aplica sobre ella la clave pública del usuario que firmó el documento.
6. El destinatario usa la misma función resumen que fue usada para firmar el documento sobre el documento recibido.
7. El destinatario comprueba que el resumen que ha obtenido del documento es igual al resumen que obtiene tras aplicar la clave pública sobre la firma. Si es así se valida la firma.

El algoritmo básico de firma (o de cifrado, ya que el principio es el mismo, salvo la clave que se usa para cifrar y descifrar) sería el siguiente:

- Se tiene una $\boldsymbol{N}$ (denominada módulo) obtenida por el producto de dos números primos enormes $\boldsymbol{p}$ y $\boldsymbol{q}$.
- Se tiene además que $\boldsymbol{\Phi(N) = (p-1)\cdot(q-1)}$. Esta es la trampa del criptosistema.

- Se elige un número $\boldsymbol{e}$ (exponente público) tal que $\mathbf{1} < e < \boldsymbol{\Phi}(\boldsymbol{N})$ y que cumpla $\boldsymbol{mcd}\left(\boldsymbol{e}, \boldsymbol{\Phi}(\boldsymbol{N})\right) = \mathbf{1}$.

- Se calcula $\boldsymbol{d}$ (exponente privado) como $\boldsymbol{e} \cdot \boldsymbol{d} \equiv \mathbf{1}\ \boldsymbol{mod}\ (\boldsymbol{\Phi}(\boldsymbol{N}))$. Aquí se ve la trampa de criptosistema. Para calcular $\boldsymbol{d}$ debo conocer $\boldsymbol{\Phi}(\boldsymbol{N})$ y sin conocer $\boldsymbol{p}$ y $\boldsymbol{q}$ es imposible.

- Se considera la clave pública el par de números $(\boldsymbol{e}, \boldsymbol{N})$ y la privada $(\boldsymbol{d}, \boldsymbol{N})$.

- Para firmar un mensaje M el proceso sería: $\boldsymbol{firma} \equiv \boldsymbol{hash}\ (\boldsymbol{M})^{\boldsymbol{e}} (\boldsymbol{mod}\ \boldsymbol{N})$

- Para comprobar la firma se usaría: $\boldsymbol{hash}\ (\boldsymbol{M}) \equiv \boldsymbol{firma}^{\boldsymbol{d}} (\boldsymbol{mod}\ \boldsymbol{N})$.

2.4 – PKCS#12

PKCS#12 es el estándar de RSA ideado para la transferencia de información de identidad de un sujeto. En otras palabras, es una forma de poder exportar, entre otros, la clave privada de una persona.

Un PKCS#12 habitualmente contiene el par de claves del usuario (la pública y la privada) y el certificado asociado a ellas. La información contendía dentro de un PKCS#12, como en otros casos, sigue una estructura ASN.1, basada en una gramática y usando reglas DER para representar la información que contiene. No se va a entrar a especificar cómo se compone, al contrario de cómo se hizo con los certificados, ya que PKCS#12 es una tecnología usada en el proyecto, no en sí una parte gestionada como era el caso de los certificados.

El PKCS#12 que contiene toda la información antes mencionada se conoce como PFX. Un PFX contiene información sensible ya que contiene la clave privada del usuario. Por tanto, debe proveerse de mecanismos que garanticen la seguridad del PFX.

El esquema básico de PFX que se usará en este proyecto (de los varios tipos que son propuestos en el estándar) es el de protección con contraseña simétrica. En otras palabras, que toda la información contenida en el PFX, especialmente la clave privada, es cifrada por medio de un algoritmo simétrico y por medio de una contraseña. Un algoritmo muy utilizado en generación de PFX, aunque no demasiado seguro, es el Triple DES.

Además de asegurar la protección de los datos contenidos en el PFX (por medio del cifrado), se debe garantizar la integridad de los mismos, ya que si en el transporte del PFX algo se modificara las claves serían inservibles. Para ello, los PFX llevan un resumen (generalmente SHA1 de 160 bits) que asegura la integridad de los datos contenidos.

Tanto para el descifrado de los datos, como para la verificación de la integridad de los mismos, el PFX lleva una serie de cabeceras (ASN.1) que describen los algoritmos y formalismos utilizados para que el PFX pueda ser utilizado posteriormente.

PKCS#12 en el ámbito de las SmartCard es muy usado para incorporar perfiles dentro de las tarjetas (incluir claves externas en la tarjeta). Esto es muy útil, ya que nos permite generar claves y certificados en una máquina y posteriormente incorporarlos a la tarjeta y poder usar la tarjeta con esa información.

Por motivos de seguridad, las claves privadas contendías en una tarjeta (ya sean generadas en ella misma o importadas a través de un PKCS#12) no pueden ser exportadas al exterior, de modo que no se puede usar PKCS#12 para extraer un perfil de una tarjeta y pasarlo a una máquina u otra tarjeta.

La codificación física más habitual de un PFX es en formato DER, que, como ya se vio en ASN.1, es una equivalencia directa del valor de los bits que forman el PFX a ASCII.

Todos los PFX usados en este proyecto han sido generados con OpenSSL a partir de claves y certificados también obtenidos con OpenSSL.

2.5 – PKCS#15

PKCS#15 es el estándar de RSA que define cómo debe ser distribuida la información dentro de una tarjeta. En otras palabras PKCS#15 es el formato del sistema de ficheros de la SmartCard.

PKCS#15 define qué elementos debe o puede haber, qué información pueden tener y cómo se puede acceder a ellos, pero no define cómo deben ser codificados en el interior de la tarjeta ni tampoco define una serie de llamadas que sirvan para gestionarlo. Por eso, cada tarjeta necesita su propia librería PKCS#11, ya que es a través de PKCS#11 la forma básica de acceder a la tarjeta (PKCS#11 si define un estándar de llamadas).

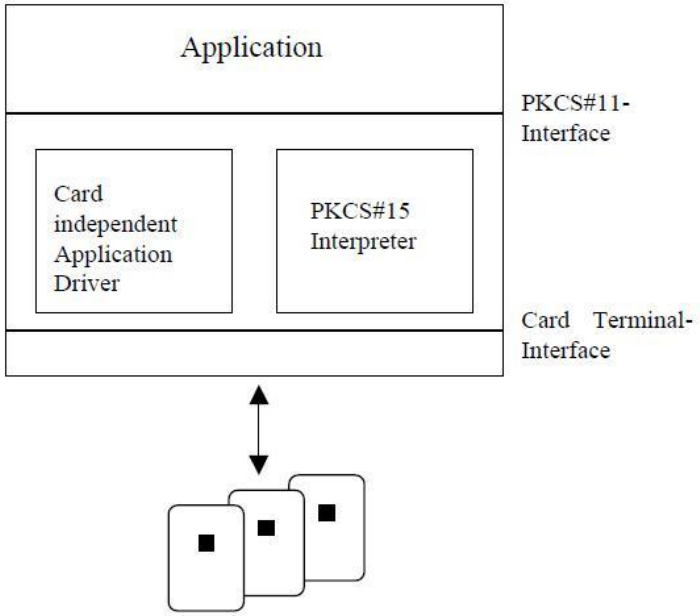

Ilustración 5: PKCS#15 como capa intermedia

En la figura se aprecia cómo una aplicación de usuario interactúa con PKCS#11, que, a su vez, tendrá que hacer uso de la implementación propia de PKCS#15 para, finalmente, poder interactuar con las tarjetas.

El sistema de ficheros de PKCS#15 emana de un MF (Master File o Fichero Maestro). Esta estructura contiene varias referencias. En primer lugar, a un DF (Dedicated File o Fichero Dedicado) que es la única obligatoria dentro del MF.

Un DF dentro de PKCS#15 tiene la propiedad de apuntar a espacio libre donde pueden ser localizados ficheros. Además, tienen la capacidad de poder contener los

denominados EF (Elementary Files o Ficheros Elementales). De forma simplificada, puede entenderse un DF como un directorio con una capacidad de almacenamiento concreta y un EF como la estructura contenedora de los ficheros físicos que contienen los datos.

El MF suele contener una referencia al EF (DIR) que es el fichero que contiene las plantillas de las solicitudes que se hacen a la capa PKCS#15. Para hacer una operación con la tarjeta, habría que obtener la plantilla adecuada para poder invocar al sistema de ficheros y, una vez se tiene la plantilla (con los datos concretos gestionados dentro de la tarjeta), se podrían realizar las operaciones.

El MF también puede contener DFs a otros elementos que no sean en sí PKCS#15 (esto deja libertad a la tarjeta para almacenar otra información propia).

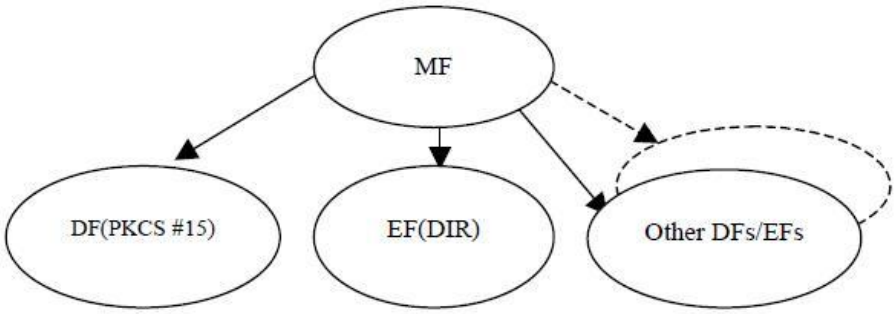

Ilustración 6: Primer nivel de PKCS#15

El DF de PKCS#15 es donde, realmente, está contenida toda la información y objetos de la tarjeta (al menos todos los que tienen que ver con PKI). El DF de PKCS#15 tiene referencias a los EF concretos que contienen todos los elementos, tales como Certificados (CDF), claves públicas, privadas y secretas (PrKDF, PuKDG, SKDF), objetos de autenticación (AODF), etc. Todos esos EF están referenciados desde un EF básico y obligatorio el EF (ODF) (Object Directory File). Este EF tiene la tabla de referencias al resto de EF (de los objetos). Serán estos EF de los objetos los que contendrán las referencias concretas a los objetos en cuestión. El DF de PKCS#15 tiene un EF (TokenInfo) que contiene información sobre el PKCS#15 de la tarjeta, capacidades, ficheros soportados, etc. Los PIN de la tarjeta (para poder acceder a ella) son almacenados en EF (AODF). Finalmente, tiene un último EF (UnusedSpace) donde se referencian las direcciones de todos los espacios contiguos no utilizados de la tarjeta, cuando se crean o eliminan objetos se modifican, añaden o eliminan entradas de este EF.

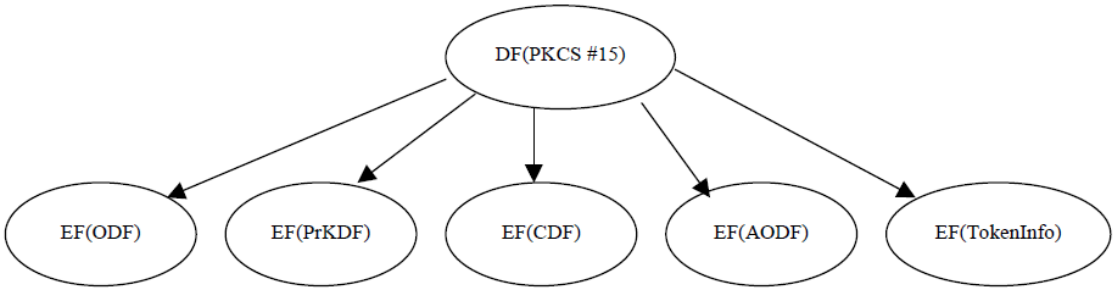

Ilustración 7: Nivel principal de PKCS#15

Por tanto, para acceder a un fichero concreto, el primer paso sería obtener la referencia del DF de PKCS#15 del MF. Luego, en el DF de PKCS#15, obtener la del EF (ODF), ahí obtener la referencia del EF del tipo de objeto concreto al que queremos acceder. Con esa referencia vamos al EF concreto y ahí se obtiene la referencia a los datos concretos que conforman un objeto. Todos los EF (que antes describimos como los ficheros) son funcionalmente muy parecidos a las referencias a datos de los i-nodos de los sistemas de ficheros EXT de sistemas Linux. En las dos siguientes ilustraciones (ejemplos de cómo se distribuyen las referencias en PKCS#15) se verá una tabla de ejemplo que representa el EF (ODF) y otra con el EF (CDF) (Fichero elemental de los certificados) donde se hace referencia real a los certificados almacenados en la tarjeta.

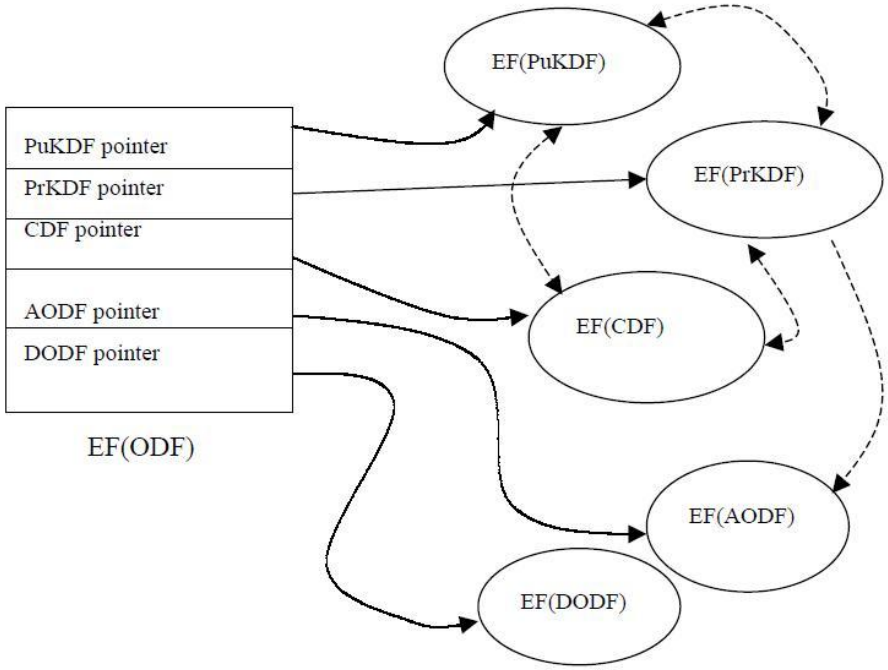

Ilustración 8: Nivel del EF(ODF)

El EF (ODF) es una tabla de punteros a todos los demás EFs (representados como bolas en la ilustración 5). Las líneas discontinuas que se ven entre ciertos elementos de los referenciados representan las relaciones existentes entre ellos, ya que toda clave pública puede tener una privada complementaria o un certificado donde esté contenida. Estas relaciones se establecen por medio de los identificadores internos.

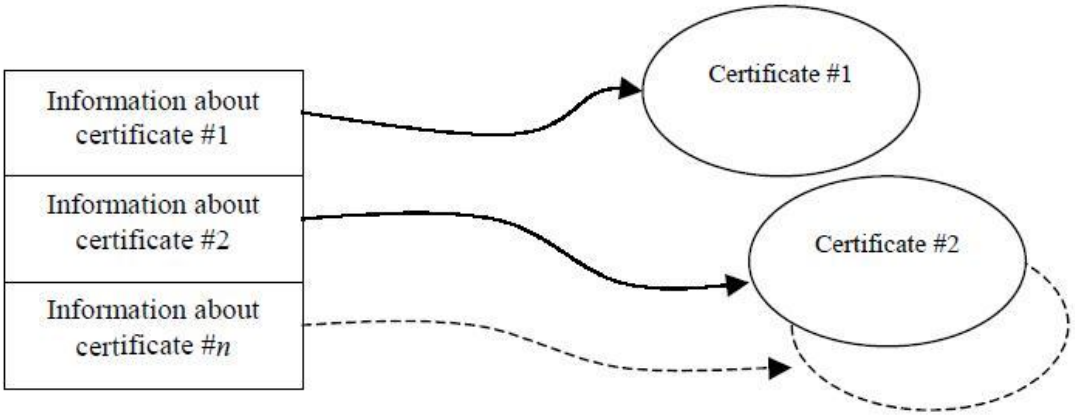

Ilustración 9: Nivel de EF(CDF) – Certificados

Aquí se aprecia como en el caso del EF (CDF) se almacenan los punteros que enlazan con los datos concretos de los certificados. Esto sería exactamente igual para el resto de objetos del sistema de ficheros con el resto de EFs.

Por tanto, PKCS#15 define la estructura de los datos dentro de la tarjeta, pero siempre será a través de PKC#11 o de llamadas no estandarizadas la forma por la cual se creen, modifiquen o eliminen los objetos.

A modo de ejemplo, y siguiendo con el caso de los certificados, si yo quisiera almacenar un certificado en la tarjeta, usando las llamadas estandarizas de PKCS#11, el proceso interno sería el siguiente:

- Se obtendría del MF la referencia del DF (PKCS#15)
- Se obtendría del DF (PKCS#15) la del EF (ODF)
- Del EF (ODF) se obtendría la del EF (CDF)
- Del EF (ODF) también se obtiene la referencia al EF (UnusedSpace)
- Se obtiene del EF (UnusedSpace) una dirección que apunte a una región, lo suficientemente grande, para el certificado que vamos a almacenar.
- Se almacena el certificado en esa área.
- Se añade la referencia en el EF (CDF) a ese certificado.

- Se modifica el EF (UnusedSpace) para que nadie almacene nada en esa área recién usada.

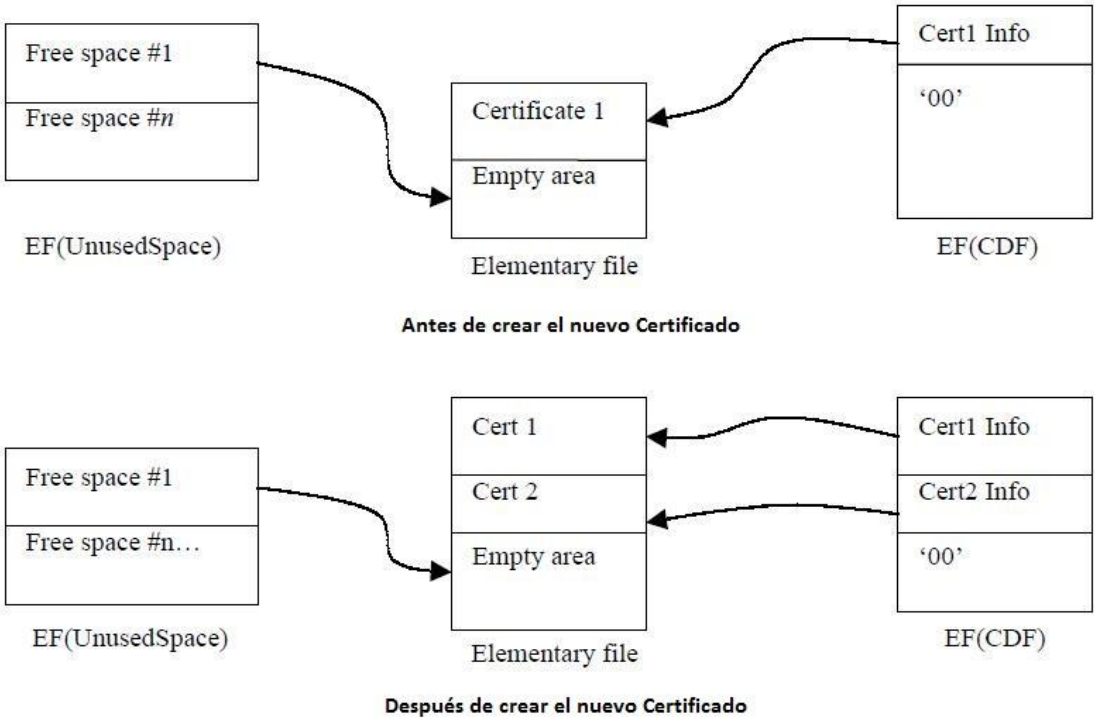

Ilustración 10: Ejemplo de creación de un certificado

2.6 – PKCS#11

PKCS#11 es el estándar RSA usado para interactuar con los tokens criptográficos. Por tanto, PKCS#11 es el primero de los estándares que se han visto hasta el momento que tiene una auténtica API con funcionalidades concretas que pueden ser invocadas por una aplicación para así realizar operaciones en una SmartCard.

PKCS#11 es conocido también como Cryptoki y por extensión a la librería que implementa la interfaz (en Windows será una DLL) también se la conoce con ese nombre.

PKCS#11 no está ideado para la gestión de todo tipo de SmartCards, sino sólo aquellas que tienen capacidades criptográficas (y generalmente de almacenamiento). PKCS#11 contempla dos visiones diferentes dentro de la criptografía: una primera, enfocada a PKI y a la firma digital (no se soporta el cifrado estándar asimétrico) y, por otro lado, la criptografía simétrica de clave secreta. En este proyecto, sólo nos centraremos en la primera parte (la de PKI), ya que, en el caso de la criptografía de clave secreta, las operaciones criptográficas no las realiza la tarjeta (ésta sólo almacena la clave) y es la librería de Cryptoki (una DLL software) la que realiza las operaciones.

Las aplicaciones que usen la librería Cryptoki realizan invocaciones por medio de una interfaz conocida (la que implementa el estándar PKCS#11). Estas librerías con una interfaz común, difieren en su implementación; ya que la organización interna de las tarjetas difiere (diferentes formas de PKCS#15, distintas codificaciones de la información interna, etc.), de modo que cada SmartCard dispone de su propia librería y no es válida para el resto. Es importante destacar que todas las librerías de Cryptoki (DLL en Windows y SO en Linux) están programadas en ANSI C.

PKCS#11 se basa en sesiones de usuarios. Esto supone que un usuario que abre la sesión tendrá acceso (en función del tipo de usuario y el tipo de sesión) a una serie de objetos dentro de la tarjeta y a poder realizar una serie de operaciones. Hay básicamente dos tipos de sesiones: las de escritura (en las que se puede modificar el contenido de la tarjeta) y las de lectura. Hay también dos tipos de usuarios: el normal y el Oficial de Seguridad (que se encarga de ciertos aspectos concretos de la gestión, como ya se verá). Los objetos tratados en las SmartCard relacionados con PKI básicamente serán clave pública, privadas y certificados.

PKCS#11 está enfocado a la gestión del contenido de las tarjetas y a la invocación de funcionalidad que es realizada con esos objetos contenidos. Deben ser aplicaciones o sistemas externos los que saquen partido de esas posibilidades. Por ejemplo, PKCS#11 no dispone de funcionalidad para resolver retos respuesta u

operaciones de autenticación. Para ello, debe haber una capa del sistema operativo, por ejemplo, que haga de intermediaria entre la tarjeta y el sistema en sí.

La API de PKCS#11 viene definida dentro de cuatro ficheros de cabecera ofrecidos por RSA. Estos ficheros son:

- criptoki.h. Encargado de definir la representación de las funciones dentro de la API y el enlazado con la librería Cryproki.
- pkcs11.h. Elemento de enlace de todas las definiciones de PKCS#11.
- pkcs11t.h. Fichero que define todos los tipos PKCS#11.
- pkcs11f.h. Fichero encargado de definir todas las cabeceras de las funciones PKCS#11.

2.6.1 – SESIONES Y USUARIOS

El primer paso para poder interactuar con una tarjeta es abrir una sesión. Hay muchas posibilidades que se irán viendo a lo largo de este punto. En primer lugar, decir que, para abrir una sesión, sólo se necesita conocer el slot donde se encuentra la SmartCard. Así que, al abrir una sesión, simplemente se tendrá acceso a todos los objetos públicos que haya en la tarjeta (en principio, claves públicas y certificados). Además, no se tendrá permiso para modificar nada en la tarjeta.

Si se abren sesiones de usuario, se requiere el conocimiento del PIN. Las sesiones de usuario dan acceso total a la tarjeta, de modo que se pueden ver objetos públicos y privados. Las claves privadas son siempre objetos privados dentro de la tarjeta, por lo que, para poder firmar un documento, hay que tener acceso a la clave privada y es necesario haber hecho login en la tarjeta con el PIN de usuario.

La seguridad de las tarjetas (ya que contienen claves privadas) al final radica en la introducción de un PIN. Este PIN en función de los tipos de tarjetas puede no ser muy extenso o demasiado seguro. Para evitar este problema y que se puedan realizar ataques de fuerza bruta para obtener el PIN de una tarjeta, éstas disponen de un número de intentos determinados (generalmente 3). Por tanto, si alguien introduce mal ese número de veces consecutivas el PIN, la tarjeta se bloquea y es imposible acceder a ella hasta que no se desbloquee.

Las sesiones pueden ser de lectura o de escritura. En las sesiones de lectura, se tiene acceso a todos los objetos de la tarjeta y a toda su funcionalidad, salvo toda la relacionada con la eliminación, creación o modificación de objetos. En las sesiones de escritura, por el contrario, sí que se tiene la posibilidad de modificar la información de la tarjeta. En este proyecto siempre se hará uso de sesiones de escritura.

En una tarjeta, las sesiones que se pueden abrir son de usuario normal o de Oficial de Seguridad. Los dos usuarios, en principio, tienen los mismos accesos a funcionalidad y objetos, salvo que el Oficial de Seguridad tiene un acceso extra a la funcionalidad relacionada con el desbloqueo del PIN de la tarjeta, etc. Por ello, el PIN del usuario y el del Oficial de Seguridad son, en principio, distintos.

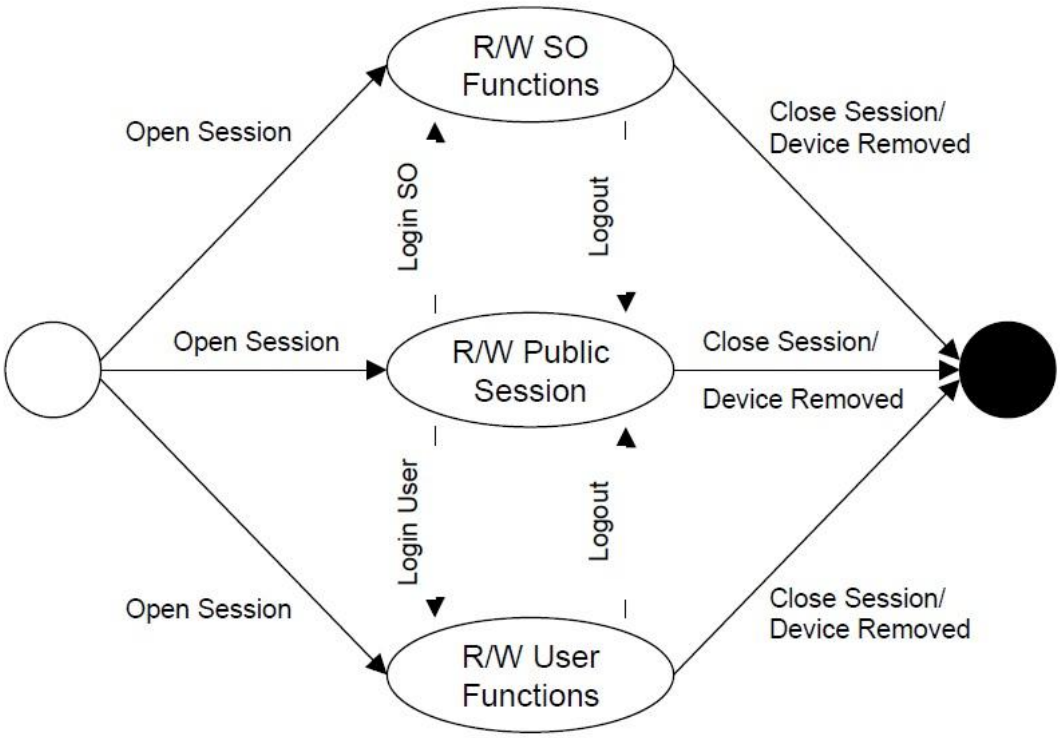

Ilustración 11: Sesiones en PKCS#11

En el diagrama se aprecian todas las transiciones permitidas en PKCS#11 entre usuarios y sesiones. En la parte superior, se ve la apertura de sesión de SO (Secure Officer u Oficial de Seguridad), abajo, la de un usuario normal y, en el centro, una sesión anónima (sin PIN).

Cada sesión tiene un identificador de sesión que la distingue del resto (y que es muy usado dentro de las llamadas de PKCS#11). Este identificador podría variar cada vez que alguien se conecta a la tarjeta (por eso debe ser recuperado en cada inicio de sesión). En caso de haber varias aplicaciones trabajando sobre la misma tarjeta, cada aplicación recibiría un identificador de sesión diferente.

Dentro de las SmartCards, no sólo se discrimina entre objetos públicos y objetos privados, sino también que existen objetos de sesión y objetos del token. Los

objetos de token son aquellos objetos persistentes, que, una vez se ha cerrado la sesión (o se ha extraído la tarjeta que a todos los efectos es cerrar la sesión), permanecen dentro del chip criptográfico. Los objetos de sesión son aquellos que existen desde que son creados hasta que se cierra la sesión, luego son eliminados de forma automática. Es más, si hubiera dos aplicaciones accediendo a la tarjeta, lo harían obligatoriamente en dos sesiones diferentes, pues bien, si en una de esas sesiones se crese un objeto de sesión, éste no podría ser visto en ninguna otra sesión.

Sobre los objetos de sesión, no hay restricción alguna sobre la escritura. De hecho, pueden escribirse, aunque no se haya abierto una sesión de escritura. Esto se debe a que estos objetos nunca están incluidos dentro de la tarjeta, sino que son gestionados en la memoria del ordenador por el Cryptoki.

Tipo de objeto	**Tipo de sesión**				
	Pública Lectura	Pública Escritura	Usuario Lectura	Usuario Escritura	Oficial de Seguridad
Públicos de Sesión	R/W	R/W	R/W	R/W	R/W
Privados de sesión			R/W	R/W	
Públicos del Token	R/O	R/W	R/O	R/W	R/W
Privados del Token			R/O	R/W	

Tabla 3: Sesiones PKCS#11

La tabla expresa sobre qué tipos de objetos tiene acceso cada usuario (y en qué régimen). Además, se ven las sesiones que puede abrir cada tipo de usuario. R/W representa sesión de Lectura/Escritura y R/O Lectura/Sólo. Se observa que la gran limitación del oficial de seguridad (no seguida siempre por las implementaciones de PKCS#11) es que sólo puede acceder a objetos públicos.

2.6.2 – TIPOS DE DATOS PKCS#11

Para poder trabajar adecuadamente en PKCS#11, hay que conocer los tipos de datos de los que dispone (generalmente obtenidos a partir de la declaración `typedef` de C) y las constantes que gestiona el estándar (principalmente sus significados).

En primer lugar, vamos a ver una tabla con los principales prefijos utilizados para describir los tipos de datos y las constantes, ya que todos los tipos en PKCS#11 se construyen a partir de esas definiciones.

Prefijo	Descripción
C_	Este es el prefijo de las funciones.
CK_	Este es el prefijo genérico de los tipos de datos y usado directamente en las constantes.
CKA_	Hace referencia a los atributos de las plantillas.
CKC_	Hace referencia al tipo de certificado.
CKF_	Hace referencia a los flags de bits
CKH_	Representa características hardware
CKK_	Representa tipos de claves
CKO_	Representa tipos de objetos
CKR_	Hace referencia valores retornados
CKU_	Representa tipos de usuarios

Tabla 4: Prefijos de PKCS#11

En la tabla, no están representados todos los prefijos de PKCS#11, pero sí lo están los de mayor importancia y todos los utilizados en el desarrollo de este proyecto. Existe un sufijo muy utilizado en PKCS#11 (el único) que está presente para casi todos los tipos de datos `_PTR` y representa un puntero al tipo de dato en cuestión. Por ejemplo, si `CK_BYTE` hace referencia a un byte `CK_BYTE_PTR` hace referencia a un puntero a un byte. El puntero nulo se presenta con `NULL_PTR`.

La siguiente tabla representa los principales tipos de datos y estructuras usadas en este proyecto:

Nombre	Tipo de dato	Valores	Uso
CK_CHAR	Carácter	Caracteres ASCII	Representación de caracteres y cadenas.
CK_BBOOL	Booleano	CK_TRUE, CK_FALSE	Representa valores verdades y falsos
CK_BYTE	Byte	Enteros 8 bits	Se usa generalmente para representar cadenas de caracteres.
CK_ULONG	Long	Enteros de 32 bits	Se usa para representar números enteros
CK_VOID	Void	Sin valor	Se usará para representar el tipo void.
CK_VERSION	Estructura	Versión del	Para almacenar el valor de la

Nombre	Tipo de dato	Valores	Uso
		Cryptoki	versión del Cryptoki
CK_INFO	Estructura	Datos sobre el Cryptoki	Representa información variada sobre el Cryptoki
CK_SLOT_ID	CK_ULONG	Enteros de 32 bits	Representa el identificador del lector de tarjetas
CK_SLOT_INFO	Estructura	Información del lector	Representa información asociada al lector en uso
CK_TOKEN_INFO	Estructura	Datos de la SmartCard	Representa datos asociados a la SmartCard (mediante flags en su mayoría)
CK_SESSION_HANDLE	CK_ULONG	Enteros 32 bits	Representa el manejador de la sesión
CK_USER_TYPE	CK_ULONG	CKU_USER, CKU_SO	Representa el tipo de usuario
CK_STATE	CK_ULONG	Estado de la sesión	Representa el estado de la sesión (mediante flags)
CK_SESSION_INFO	Estructura	Información de la Sesión	Representa datos asociados a la sesión (mediante flags)
CK_OBJCT_HANDLE	CK_ULONG	Enteros de 32 bits	Representa el manejador de un objeto
CK_OBJECT_CLASS	CK_ULONG	Enteros de 32 bits	Representa el tipo de objetos (certificados, claves, etc.)
CK_KEY_TYPE	CK_ULONG	Enteros de 32 bits	Representa el tipo de clave
CK_CERTIFICATE_TYPE	CK_ULONG	Enteros de 32 bits	Representa el tipo de certificado
CK_ATTRIBUTE_TYPE	CK_ULONG	Enteros de 32 bits	Representa tipos de atributos
CK_ATTRIBUTE	Estructura	Tipos de atributos y sus valores	Almacena atributos y sus correspondientes valores
CK_DATE	Estructura	Fechas	Almacena fechas
CK_MECHANISM_TYPE	CK_ULONG	Enteros de 32 bits	Representa tipos de mecanismos
CL_MECHANISM	Estructura	Tipos de mecanismos y sus valores	Almacena mecanismos y sus correspondientes valores.
CK_MECHANISM_INFO	Estructura	Datos de los mecanismos	Representa datos asociados a los mecanismos (mediante el uso de flags).
CK_RV	CK_ULONG	Enteros de 32 bits	Representa el tipo de retorno de todas las funciones del Cryptoki

Tabla 5: Tipos de datos PKCS#11

Aunque los tipos básicos de PKCS#11 y los miembros que componen las estructuras pueden sustituirse por tipos básicos de C sin perder la compatibilidad, es mucho más sencillo trabajar con los tipos redenominados de PKCS#11, debido a que en las especificaciones todo viene tratado con dichos tipos y, si se usaran los tipos básicos, habría que memorizar qué tipo corresponde con qué redenominación.

Todos los tipso de datos tratados en PKCS#11 vienen definidos en el fichero de cabecera pkcs11t.h.

2.6.3 – OBJECTOS PKCS#11

Los objetos de PKCS#11 son las estructuras de datos gestionadas en las tarjetas (dentro de estructuras PKCS#15) que se corresponden con elementos (generalmente PKI) que pueden existir fuera de la tarjeta, como son los certificados o las claves.

Los objetos en PKCS#11 se presentan como estructuras de datos que heredan, de su estructura padre, todos los atributos que contiene y añaden nuevos atributos necesarios para el tipo de objeto que representan.

Todos los objetos emanan del tipo Object y los más importantes (y los que conciernen a PKI) emanan de Storable Object. Los más destacados son los certificados, las claves públicas y las claves privadas. Por tanto, será en la línea Object → Object Storable → Objetos certificado y claves, en la que nos centraremos.

En este proyecto, se gestionan tres tipos concretos de objetos: los certificados X.509, las claves públicas y las claves privadas (todos de tipo Storage). Estos objetos tienen básicamente dos atributos (heredados de Storage) que son importantes más allá de los datos concretos del objeto en sí. Son los Atributos `CKA_TOKEN` y `CKA_PRIVATE`. El atributo `CKA_TOKEN` con valor `CK_TRUE` expresa que el objeto será un objeto de token, o, lo que es lo mismo, perdurable. En caso de ser `CK_FALSE` el objeto en cuestión, lo será de sesión y desaparecerá al cerrarse la sesión. `CKA_PRIVATE` nos indica si el objeto en cuestión es o no privado. Privado supone que sólo un usuario que introduzca el PIN de la tarjeta podrá tener acceso a dicho objeto. `CKA_PRIVATE` con valor `CK_TRUE` será privado y con valor `CK_FALSE` será público. Por defecto los objetos son públicos, salvo las claves privadas que son objetos privados. Además, es importante CKA_EXTRACTABLE que nos dice si el objeto puede ser extraído de la tarjeta. Los únicos objetos prohibidos de extraer, a priori, son las claves privadas.

A continuación, se expone una ilustración con los tipos principales de objetos y sus atributos, donde se ven las relaciones de herencia (un objeto que hereda de otro tiene todos los atributos contenidos en el padre):

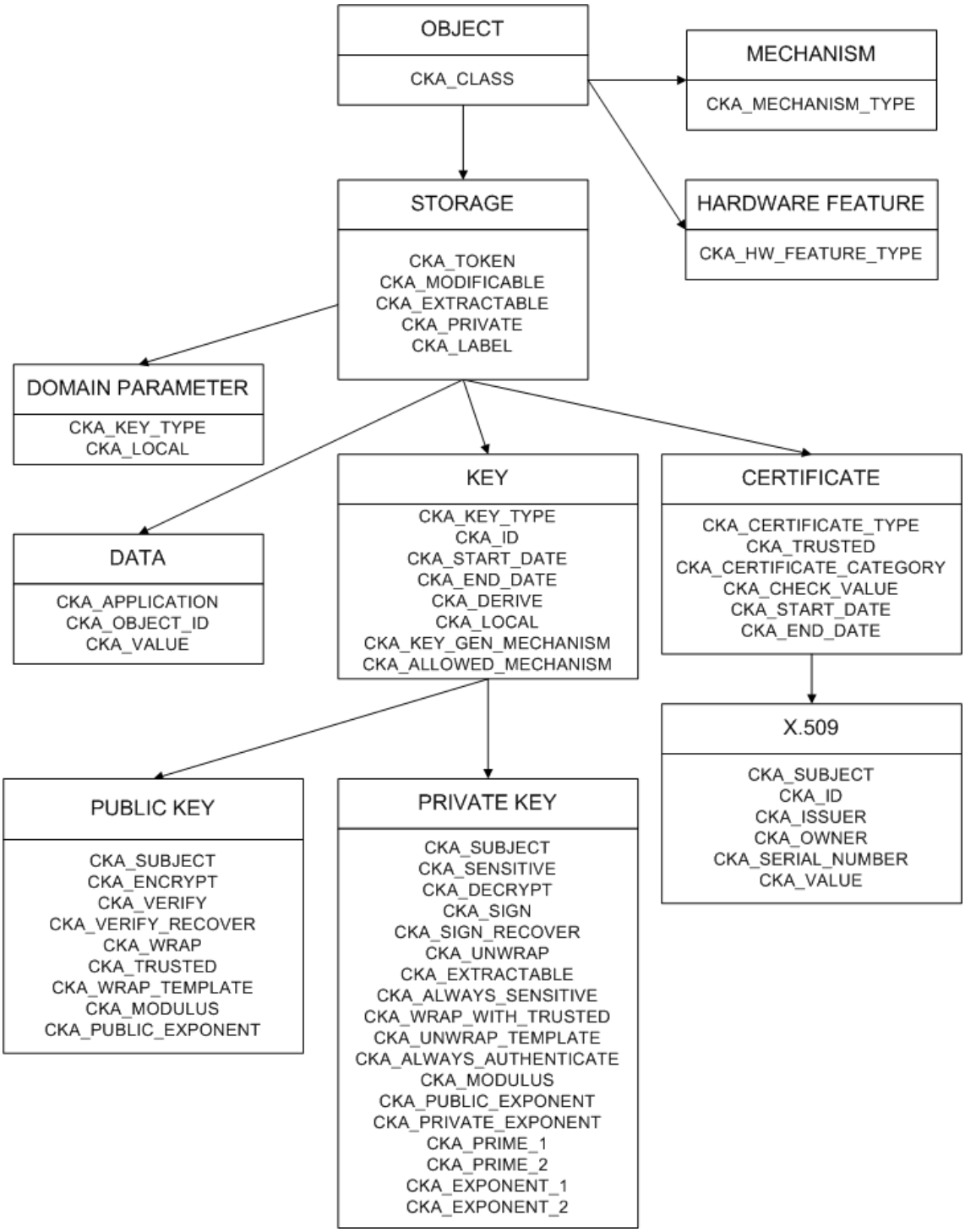

Ilustración 12: Objetos PKCS#11

Para gestionar estos objetos dentro de una aplicación PKCS#11, se hace a través de lo que se denominan templates o plantillas. Un plantilla no es más que un array que contiene tripletes de atributos, valor y longitud en bytes del valor. Esa plantilla se rellena con los datos concretos (o nulos si se espera que sea la SmartCard la

que les asigne valor) y se suministra a la tarjeta para que genere un objeto o bien para que modifique uno existente.

A continuación, vamos a exponer el uso y valores de los ditintos atributos expresados en la anterior ilustración (sólo los de tipo Storage y sus padres).

Atributo	Objeto base	Tipo	Valores	Descripción
CKA_CLASS	Object	CK_OBJECT_CLASS	CKO_DATA, CKO_CERTIFICATE, CKO_PUBLIC_KEY, CKO_PRIVATE_KEY, CKO_SECRET_KEY, CKO_HW_FEATURE, CKO_DOMAIN_ PARAMETERS, CKO_MECHANISM	Describe el tipo de objeto que tratamos (asignándole cada una de las constantes de la izquierda).
CKA_TOKEN	Storage	CK_BBOOL	CK_TRUE, CK_FALSE	Si el valor es true, el objeto es del token y, si es false, es de la sesión.
CKA_MODIFICABLE	Storage	CK_BBOOL	CK_TRUE, CK_FALSE	Si el valor es true, el objeto se puede modificar, si es false, no.
CKA_EXTRACTABLE	Storage	CK_BBOOL	CK_TRUE, CK_FALSE	Si el valor es true, el objeto se puede extraer del token, si es false no.
CKA_PRIVATE	Storage	CK_BBOOL	CK_TRUE, CK_FALSE	Si su valor es true, el objeto es privado (no visible sin PIN), si es false, el objeto es público.
CKA_LABEL	Storage	CK_BYTE_PTR	Cadena de caracteres	Es la etiqueta del objeto.
CKA_CERTIFICATE_ TYPE	Certificate	CK_CERTIFICATE_ TYPE	CKC_X_509, CKC_X_509_ATTR_CERT, CKC_WTLS, CKC_VENDOR_DEFINED	Define el tipo de certificado en cuestión. Sólo se usa CKC_X_509
CKA_TRUSTED	Certificate	CK_BBOOL	CK_TRUE, CK_FALSE	Define si el certificado es o no de confianza.
CKA_CERTIFICATE_ CATEGORY	Certificate	CK_ULONG	0, 1, 2, 3	Tipo de categoría, 0 es sin especificar, 1 es token de usuario, 2 autoridad y 3 otra entidad.
CKA_CHECK_ VALUE	Certificate	CK_BYTE_PTR	Cadena de bytes	Suma de seguridad.
CKA_START_DATE	Certificate	CK_DATE	YYYY/MM/DD	La fecha de inicio de validez del certificado
CKA_END_DATE	Certificate	CK_DATE	YYY/MM/DD	La fecha de fin de validez del certificado
CKA_SUBJECT	X.509	CK_BYTE_PTR	DER	Codificación DER del nombre del sujeto
CKA_ID	X.509	CK_BYTE_PTR	Cadena de bytes	Es el identificador que relaciona los objetos de la tarjeta.
CKA_ISSUER	X.509	CK_BYTE_PTR	DER	Codificación DER del

Atributo	Objeto base	Tipo	Valores	Descripción
				emisor del certificado.
CKA_OWNER	X.509	CK_BYTE_PTR	DER	Codificación DER del propietario del sujeto.
CKA_SERIAL_ NUMBER	X.509	CK_BYTE_PTR	Cadena de bytes	Es el Número de Serie del certificado.
CKA_VALUE	X.509	CK_BYTE_PTR	DER	Es la codificación DER completa del certificado.
CKA_KEY_TYPE	Key	CK_KEY_TYPE	DKK_RSA, DKK_DSA, DKK_DH	Define el tipo de clave.
CKA_ID	Key	CK_BYTE_PTR	Cadena de bytes	Es el identificador que relaciona los objetos de la tarjeta.
CKA_START_DATE	Key	CK_DATE	YYYY/MM/DD	La fecha de inicio de validez del certificado
CKA_END_DATE	Key	CK_DATE	YYYY/MM/DD	La fecha de fin de validez del certificado
CKA_DERIVE	Key	CK_BBOOL	CK_TRUE, CK_FALSE	True si la clave permite derivación de terceras. False en caso contrario.
CKA_LOCAL	Key	CK_BBOOL	CK_TRUE, CK_FALSE	True si sólo se permite la creación. False si también se permite la copia.
CKA_KEY_GEN_ MECHANISM	Key	CK_MECHANISM_ TYPE	CKK_RSA_PKCS_KEY_ PAIR_GEN	Mecanismo de generación de las claves.
CKA_ALLOWED_ MECHANISM	Key	CK_MECHANISM_ TYPE_PTR	Lista de mecanismos	Lista de los mecanismo válidos para usar con esta clave.
CKA_SUBJECT	PuKey	CK_BYTE_PTR	DER	Codificación DER del propietario de la clave.
CKA_ENCRYPT	PuKey	CK_BBOOL	CK_TRUE, CK_FALSE	Si su valor es true, la clave sirve para encriptar. Si es false, no.
CKA_VERIFY	PuKey	CK_BBOOL	CK_TRUE, CK_FALSE	Si su valor es true, la clave sirve para verificar firmas. Si es false, no.
CKA_VERIFY_ RECOVER	PuKey	CK_BBOOL	CK_TRUE, CK_FALSE	Si su valor es true, la clave sirve para verificar firmas en varias partes. Si es false, no.
CKA_WRAP	PuKey	CK_BBOOL	CK_TRUE, CK_FALSE	Si su valor es true, la clave sirve para cifrar otras claves. Si es false, no.
CKA_TRUSTED	PuKey	CK_BBOOL	CK_TRUE, CK_FALSE	Si su valor es true, la clave es de confianza. Si es false, no.
CKA_WRAP_ TEMPLATE	PuKey	CK_ATTRIBUTE_ PTR	Lista de atributos	Plantilla a utilizar para codificar una clave.
CKA_MODULUS	PuKey	CK_BYTE_PTR	Cadena de bytes	Este es el módulo de la clave pública.
CKA_PUBLIC_	PuKey	CK_BYTE_PTR	Cadena de bytes	Este es el exponente de la clave pública.

Atributo	Objeto base	Tipo	Valores	Descripción
EXPONENT				
CKA_SUBJECT	PrKey	CK_BYTE_PTR	DER	Codificación DER del propietario de la clave.
CKA_SENSITIVE	PrKey	CK_BBOOL	CK_TRUE, CK_FALSE	Si es true la clave sólo puede ser gestionada por el Hardware de la tarjeta. Siempre es true.
CKA_DECRIPT	PrKey	CK_BBOOL	CK_TRUE, CK_FALSE	Si es true, la clave sirve para desencriptar datos. Si es false, no.
CKA_SIGN	PrKey	CK_BBOOL	CK_TRUE, CK_FALSE	Si es true, la clave sirve para firmar datos. Si es false, no.
CKA_SIGN_ RECOVER	PrKey	CK_BBOOL	CK_TRUE, CK_FALSE	Si es true, la clave sirve para firmar datos recuperados de la firma.
CKA_UNWRAPED	PrKey	CK_BBOOL	CK_TRUE, CK_FALSE	Si es true, la clave sirve para decodificar claves. Si es false, no.
CKA_STRACTABLE	PrKey	CK_BBOOL	CK_TRUE, CK_FALSE	Si es true, la clave puede ser extraída de la tarjeta. Siempre es false.
CKA_ALWAYS_ SENSITIVE	PrKey	CK_BBOOL	CK_TRUE, CK_FALSE	Siempre es true, ya que CK_SENSITIVE siempre lo es.
CKA_WRAP_ WHIT_TRUSTED	PrKey	CK_BBOOL	CK_TRUE, CK_FALSE	Si es true, la clave sirve para cifrar sólo claves de confianza. Si es false, puede cifrar cualquiera.
CKA_UNWRAP_ TEMPLATE	PrKey	CK_ATTRIBUTE_ PTR	Lista de atributos	Plantilla a utilizar para decodificar una clave.
CKA_ALWAYS_ AUTHENTICATE	PrKey	CK_BBOOL	CK_TRUE, CK_FALSE	Si es true, el PIN debe ser suministrado en cada uno.
CKA_MODULUS	PrKey	CK_BYTE_PTR	Cadena de bytes	Este es el módulo de la clave.
CKA_PUBLIC_ EXPONENT	PrKey	CK_BYTE_PTR	Cadena de bytes	Este es el exponente público.
CKA_PRIVATE_ EXPONENT	PrKey	CK_BYTE_PTR	Cadena de bytes	Este es el exponente privado.
CKA_PRIME_1	PrKey	CK_BYTE_PTR	Cadena de bytes	Es el primo p de RSA.
CKA_PRIME_2	PrKey	CK_BYTE_PTR	Cadena de bytes	Es el primo q de RSA.
CKA_EXPONENT_1	PrKey	CK_BYTE_PTR	Cadena de bytes	Exponente interno de RSA.
CKA_EXPONENT_2	PrKey	CK_BYTE_PTR	Cadena de bytes	Exponente interno de RSA.

Tabla 6: Atributos de los objetos PKCS#11

Una plantilla de un objeto, ya sea para crearlo o modificarlo, no tiene por qué tener todos los atributos correspondientes a ese tipo de objeto (más los heredados).

Generalmente todos los atributos tienen valores, por defecto, prefijados por el Cryptoki concreto que se está usando, de modo que los objetos que se crean sean correctos sin tener que especificar valor a todos los atributos. El problema es que esto no se cumple, por completo, en las implementaciones reales de las distintas librerías PKCS#11. Hay librerías que tienen valores por defecto no adecuados (sin inicializar), de modo que si no se especifican ciertos atributos será imposible gestionar el objeto, mientras que en otra librería de otra tarjeta el objeto se puede gestionar sin problemas sin especificar dicho atributo.

Además del problema de los atributos y sus valores por defecto, cuando no son expresados en la plantilla existe el problema de qué atributos concretos deben ir en la plantilla. Una misma plantilla con dos Cryptoki diferentes para dos tarjetas diferentes puede funcionar en uno y fallar en otro, por lo que es tremendamente difícil encontrar plantillas compatibles para todos los tipos de tarjetas. De hecho, en este proyecto se han utilizado varios tipos de plantillas para la gestión de un mismo tipo de objeto en función de la tarjeta que se fuera a utilizar. En apartados posteriores del proyecto, relacionados con la implementación, se definirán con el código las plantillas utilizadas para la realización del presente proyecto.

Todos los objetos, dentro de una tarjeta, tienen un identificador asignado que los diferencia unívocamente. Estos identificadores son los denominados manejadores y nos permiten referenciar a un objeto para poder utilizarlos, modificarlos, eliminarlos, etc. Es más, al crear los objetos siempre recibimos su manejador por si es necesario utilizarlo. Estos manejadores son diferentes en cada sesión, de modo que un mismo objeto puede tener (y de hecho tendrá) dos manejadores distintos en dos sesiones diferentes.

2.6.4 – FUNCIONES PKCS#11

Para interactuar con PKCS#11 se deben utilizar las funciones que provee la API. Es la única forma de utilizar la tarjeta, ya sea para abrir sesiones, gestionar objetos, firmar, etc. Todo se hace a través de invocaciones a las funciones en el orden adecuado y pasando los parámetros necesarios.

Las funciones PKCS#11 son, en general, bastante intuitivas, aunque para realizar operaciones relativamente sencillas, como puede ser listar los objetos de la tarjeta, hay que involucrar gran número de invocaciones diferentes.

Todas las funciones de PKCS#11 retornan un `CK_RV` con el resultado de la invocación. Si todo ha ido bien, retornarán `CKR_OK` (la función se ejecutó correctamente). En caso de haber algún error, retornarán otro valor descriptivo que

permita entender por qué no se realizó bien la ejecución de la función. Por esta forma de crear las funciones, cuando queremos recibir de una función alguna información necesaria (como podría ser el manejador de un objeto), debemos hacerlo a través de paso de parámetros por referencia. Las funciones de PKCS#11 están llenas de parámetros por referencia.

En las invocaciones de las funciones, en general, hay tres argumentos que son constantes en casi todas ellas. El primero, es el `CK_SESSION_HANDLE` que nos representa el manejador de la sesión actual (sirve en el Cryptoki para discriminar entre varias sesiones abiertas). El segundo, es el `CK_SLOT_ID` que representa el lector en cuestión que se está utilizando (podríamos gestionar en varios lectores de forma simultánea). El último, sería el `CK_OBJECT_HANDLE` que sería el manejador de alguno de los objetos que hay en la tarjeta (como ya se dijo, para todo lo relacionado con la gestión de los objetos, se utiliza su manejador concreto).

El Cryptoki reparte las 68 funciones soportadas en 13 categorías distintas en función de la finalidad de las mismas. A continuación, se va a exponer una tabla con todas las funciones soportadas y una breve descripción de su uso.

Categoría	Función	Descripción
Funciones de propósito General	C_Initialize	Inicializa el Cryptoki (para poder ser usada la librería).
	C_Finalize	Libera el uso del Cryptoki (para cuando ya no se quiere usar más la librería)
	C_GetInfo	Obtiene información sobre el Cryptoki que se está usando.
	C_GetFunctionList	Obtiene punteros a las funciones del Cryptoki (para poder invocarlas).
Funciones para el manejo de los Slot y los Tokens	C_GetSlotList	Esta función nos da una lista de lectores en el sistema.
	C_GetSlotInfo	Da información acerca de un lector concreto.
	C_GetTokenInfo	Obtiene información del token que está en un lector determinado.
	C_WaitForSlotEvent	Esta función espera a que se produzca un evento en el lector (como meter o sacar una tarjeta).
	C_GetMechanismList	Obtiene una lista de los mecanismos soportados por la tarjeta.
	C_GetMachanismInfo	Obtiene información de un mecanismo concreto (por medio de flags).
	C_InitToken	Inicializa una tarjeta (la formatea al estado de fábrica). Sólo se puede usar en tarjetas de desarrollador.
	C_InitPIN	Inicializa un PIN de usuario bloqueado.

Categoría	Función	Descripción
	C_SetPIN	Establece un nuevo PIN de usuario
Funciones del manejo de la sesión	C_OpenSession	Abre una sesión dentro de la tarjeta.
	C_CloseSession	Cierra una sesión abierta en la tarjeta.
	C_CloseAllSessions	Cierra todas las sesiones que estuvieran abiertas en la tarjeta.
	C_GetSessionInfo	Obtiene información de la sesión (por medio de flags).
	C_GetOperationState	Obtiene información de la operación que se esté realizando.
	C_SetOperationState	Establece un estado de operación.
	C_Login	Autentica a un usuario en la tarjeta.
	C_Logout	Hace logout del usuario en la tarjeta.
Funciones para el manejo de objetos	C_CreateObject	Crea un nuevo objeto en la tarjeta.
	C_CopyObject	Copia un objeto de la tarjeta a otro objeto nuevo.
	C_DestroyObject	Elimina un objeto de la tarjeta.
	C_GetObjectSize	Obtiene el tamaño en bytes de un objeto dentro de la tarjeta.
	C_GetAttributeValue	Obtiene el valor de un atributo (o lista de ellos) de un objeto.
	C_SetAttributeValue	Establece nuevos valores para un atributo (o lista de ellos) para un objeto.
	C_FindObjectsInit	Inicializa la búsqueda de objetos en la tarjeta. Se pueden usar plantillas para discriminar.
	C_FindObjects	Busca objetos dentro de una búsqueda inicializada.
	C_FindObjectsFinal	Finaliza una búsqueda previamente inicializada.
Funciones de encriptación	C_EncryptInit	Inicializa una encriptación.
	C_Encrypt	Encripta datos de una sola vez.
	C_EncryptUpdate	Actualiza una encriptación ya inicializada.
	C_EncryptFinal	Finaliza una encriptación previamente inicializada.
Funciones de desencriptación	C_DecryptInit	Inicializa una desencriptación.
	C_Decrypt	Desencripta datos de una sola vez.
	C_DecryptUpdate	Actualiza una desencriptación ya inicializada.
	C_DeryptFinal	Finaliza una desencriptación previamente inicializada.
Funciones de resúmenes de mensajes	C_DigestInit	Inicializa un resumen.
	C_Digest	Realiza un resumen de una vez.
	C_DigestUpdate	Actualiza un resumen ya inicializado.
	C_DigestFinal	Finaliza un resumen inicializado.
	C_DigestKey	Obtiene el resumen de una clave.
Funciones de firma	C_SingInit	Inicializa una firma.
	C_Sign	Firma datos de una sola vez.

Categoría	Función	Descripción
	C_SignUpdate	Actualiza una firma ya inicializada.
	C_SignFinal	Finaliza una firma previamente inicializada.
	C_SignRecoverInit	Inicializa una firma donde los datos son recogidos de la firma
	C_SignRecover	Realiza una firma donde los datos pueden ser obtenidos de la firma
Funciones de verificación de firmas	C_VerifyInit	Inicializa una verificación de firma.
	C_Verify	Verifica una firma de una sola vez.
	C_VerifyUpdate	Actualiza una firma ya inicializada.
	C_VerifyFinal	Finaliza una firma previamente inicializada.
	C_VerifyRecoverInit	Inicializa una verificación de firma donde los datos son recogidos de la firma
	C_VerifyRecover	Realiza una verificación de firma donde los datos pueden ser obtenidos de la firma
Funciones criptográficas de doble propósito	C_DigestEncryptUpdate	Continúa, de forma simultánea, las operaciones de resumen y encriptación.
	C_DecryptDigestUpdate	Continúa, de forma simultánea, las operaciones de desencriptación y resumen.
	C_SignEncryptUpdate	Continúa, de forma simultánea, las operaciones de firma y encriptación.
	C_DecryptVerifyUpdate	Continúa, de forma simultánea, las operaciones de desencriptación y verifiación.
Funciones para la gestión de claves	C_GenerateKey	Esta función genera una clave secreta.
	C_GenerateKeyPair	Esta función genera un par de claves (pública y privada).
	C_WrapKey	Esta función encripta una clave.
	C_UnwrapKey	Esta función desencripta una clave.
	C_DeriveKey	Esta función deriva una clave desde una clave base.
Funciones para la generación de números aleatorios	C_SeedRandom	Esta función selecciona una semilla para la generación de número aleatorios.
	C_GenerateRandom	Esta función genera números pseudoaleatorios.
Funciones para la gestión de operaciones paralelas	C_GetFunctionStatus	Obtiene el estado de una ejecución.
	C_CancelFunction	Cancela la ejecución de una función.

Tabla 7: Descripción de las funciones PKCS#11

Las funciones de los bloques relacionados con encriptación y desencriptación, así como las relacionadas con los resúmenes, no se han utilizado en este proyecto (se implementaron la realización y verificación de resúmenes, pero por ser una operación

que realiza la DLL de Cryptoki y no el hardware de la tarjeta se dejó como comentario de código). El resto de funcionalidades de PKCS#11 si que ha sido tratado en mayor o menor medida en este proyecto.

Las funciones Init, Update y Final que están presentes en muchas operaciones deben ser llamadas, una tras otra, para que la operación a la que hacen referencia pueda realizarse de forma satisfactoria (como podría ser la firma, por ejemplo). Si no se realizaran esas invocaciones de forma consecutiva no se obtendrían resultados. La finalidad de la existencia de estas funciones, cuando también existen sus equivalentes que lo hacen todo con una sola llamada, es la memoria RAM. Hacer una firma de un fichero de 8 GB (una imagen de un DVD doble capa por ejemplo) sería imposible, ya que habría que pasarle a la función un buffer con ese fichero a la función y sería excesivamente grande. Sin embargo, se podría hacer un Init, luego una serie repetida de Updates con pequeños fragmentos del fichero y, finalmente, un Final. En este proyecto, siempre se ha utilizado esta fórmula y nunca la realización de la operación con una incoación.

2.6.5 – UTILIZACIÓN DE PKCS#11

La mejor forma de entender un escenario típico de utilización de PKCS#11 es a través de un ejemplo ilustrativo que ponga de manifiesto el orden establecido de las operaciones para poder obtener resultados usando las tarjetas. Este ejemplo es extensible a otras muchas funcionalidades y en él se ven los principales pasos que hay que dar para comenzar a utilizar una tarjeta:

1. Se debe inicializar el Cryptoki. Para ello, se invoca a `C_Initialize`.
2. Debemos obtener el `CK_SLOT_ID` (identificador del lector) de la lista de lectores del sistema. Para ello, se invoca a `C_GetSlotList`.
3. Se debe abrir una sesión en el sistema. Para ello, debemos pasarle el `CK_SLOT_ID` obtenido en el paso 2 y los flags `CKF_SERIAL_SESSION | CKF_RW_SESSION`. El primero es obligado por el Cryptoki, el segundo es para sesiones de lectura/escritura. Se llamará a `C_OpenSession` para conseguirlo. En esta llamada obtenemos el `CK_SESSION_HANDLE` de la sesión.
4. Hacemos login en la tarjeta con el tipo de usuario `CKU_USER` (usuario normal) usando la función `C_Login` con el `CK_SESSION_HANDLE` antes obtenido. Con ello, se tiene acceso a objetos privados.
5. Se preparan unas plantillas de clave pública y privada compatibles con el Cryptoki que se esté utilizando y se invoca a la función

`C_GenerateKeyPair`. Con esto, se generarían un par de claves dentro de la tarjeta y obtendríamos el `CK_OBJECT_HANDLE` de cada una de las claves.

6. Con el `CK_OBJECT_HANDLE` de la clave privada, el `CK_SESSION_HANDLE` y el mecanismo adecuado invocamos `C_SignInit` para firmar un buffer de datos.
7. Hacemos sucesivas invocaciones a `C_SignUpdate` con el `CK_SESSION_HANDLE` hasta que no haya más datos que firmar.
8. Finalmente, con el `CK_SESSION_HANDLE` y la invocación a `C_SignFinal` obtenemos el resultado de la firma del buffer de datos que queríamos firmar.
9. Con el `CK_SESSION_HANDLE` y la llamada a `C_CloseSession` cerramos la sesión actual.
10. Finalmente, se invoca `C_Finalize` para restablecer el Cryptoki.

Esto sería un escenario básico donde se accede a la tarjeta, se generan un par de claves, se usa la privada para firmar unos datos y, finalmente, se sale del sistema. Los puntos del 1 al 4 serían idénticos para cualquier uso de SmartCards, al igual que los dos últimos. Los puntos que diferirían serían los comprendidos entre el 5 y el 8, que dependerían de las operaciones que quisiéramos realizar.

Como se puede observar, la práctica totalidad de las operaciones lleva como parámetro `CK_SESSION_HANDLE`, que es el manejador de la sesión. Esto es así para que el Cryptoki pueda identificar en todo momento con qué sesión está asociada cada operación, ya que el Cryptoki puede tener varias sesiones distintas abiertas.

2.7 – OPENSSL

OpenSSL es un proyecto libre con implementaciones para Linux y Windows, cuya funcionalidad principal es la gestión de arquitecturas PKI. OpenSSL tiene dos perspectivas; por una lado, el uso del repertorio de aplicaciones que tiene y, por otro, su API para invocar funcionalidades propias de PKI. En este proyecto, se hará uso de ambas, ya que será necesario el uso de las aplicaciones para gestionar la CA que se usará para firmar las peticiones de los certificados emitidos por las tarjetas, para generar archivos PFX de PKCS#12 y funcionalidades similares. Por otro lado, será necesario usar la API de OpenSSL para gestionar dentro de la aplicación desarrollada.

La principal aplicación que se va a usar de las del paquete OpenSSL será openssl.exe, que se encuentra en el /bin del directorio de instalación. Esta aplicación permite generar CAs, peticiones de certificados, hacer uso de la CA para firmar las peticiones y generar claves, generar PFXs, así como cualquier otra funcionalidad relacionada con arquitecturas PKI (revocaciones, etc.).

La API de OpenSSL que se va a utilizar está implementada dentro de la DLL (en Linux un .so) llamada libeay32.dll (que puede estar en C:\Windows\System32\ o en el directorio de instalación de OpenSSL). De esta API se hará uso de cuatro funcionalidades básicas:

- Generar CSR a partir de las claves públicas. Para ello, se hará uso de los siguientes tipos de datos:
  - `X509_REQ`. Representa la petición PKCS#10 básica.
  - `EVP_PKEY`. Es la representación de las claves, tanto públicas como privadas.
  - `BIO`. Son los objetos abstractos de entrada/salida (usados, por ejemplo, para escribir la petición en un fichero).
  - `X509_NAME`. Contiene datos personales sobre el propietario de la petición
  - `X509_EXTENSIONS`. Contiene datos sobre los usos y propiedades de la petición.

    Y las siguientes funciones:
  - `X509_REQ_new ()`. Genera una petición CSR.
  - `X509_REQ_set_pubkey ()`. Añade una clave pública a la petición.
  - `X509V3_EXT_conf ()`. Crea una neuva propeidad para el certificado.
  - `sk_X509_EXTENSION_push ()`. Añade una nueva propeidad al CSR.
  - `BIO_write ()`. Función encargada de la escritura de los objetos.

  - `*_free ()`. Funciones encargadas de liberar memoria de los objetos creados. El * representa que pueden ir multitudes de nombres.
- Importar certificados. Para ello se hará uso de los siguientes objetos
  - `X509`. Representa un certificado X509.
  - `BIO`. Ya descrito.

    Y las siguientes funciones:
  - `PEM_read_bio_X509 ()`. Es la función BIO de lectura de certificados.
  - `*_free ()`. Ya descritas.
- Importar identidades digitales desde PFX. Sus tipos de datos principales son:
  - `PKCS12`. Representa el perfil PFCS#12.
  - `BIO`. Ya descrito.
  - `EVP_PKEY`. Ya descrito.

    Las principales funciones son:
  - `BIO_read_filename ()`. Usada para leer ficheros con objetos BIO de forma genérica.
  - `d2i_PKCS12_bio ()`. Usada para convertir lo leido de un objeto BIO en un objeto PKCS#12.
  - `PKCS12_parse ()`. Usada para leer los elementos contenidos dentro de un PKCS#12 (certificado, clave privada y clave pública).
  - `*_free ()`. Ya descritas.
- Lectura segura de parámetros. Esta funcionalidad es usada para poder insertar claves en la consola de forma segura (que no sean visibles, que no se alanceen en memoria, etc.). Sus principales funciones:
  - `EVP_read_pw_string ()`. Esta función es usada para recoger los datos insertados por consola de forma privada.

En siguientes apartados (concretamente en el de punto 5 de Diseño), se profundizará mucho en cómo se ha utilizado la API de OpenSSL para dar soporte a la funcionalidad requerida (enfocada siempre a la gestión de objetos PKI).

OpenSSL también provee otras funcionalidades como la de utilizar sockets seguros para comunicación de datos por internet, protocolo HTTPS, etc.; pero, debido a que estas funcionalidades no tienen relación con la parte de PKI que se trata en este proyecto, no se verán ni serán tenidas en cuenta.

3 – SMARTCARDS

Como ya se mencionó con anterioridad, una SmartCard es una tarjeta que tiene capacidad de almacenamiento o capacidad de procesamiento (generalmente se tienen ambas). Las SmartCards, en las que nos vamos a centrar en este proyecto, son aquellas que tienen capacidad criptográfica, más concretamente las que tienen capacidad PKI.

A continuación, se van a expresar las características más importantes de los diferentes tipos de tarjetas probadas en el proyecto. Las tarjetas en cuestión son:

- Tarjeta Ceres. Es una tarjeta de identificación electrónica predecesora del DNIe.
- Starcos 2.3. Es una tarjeta convencional de usuario (no reinicializable).
- Aladdin. Es, al igual que Starcos, una SmartCard convencional; pero, al menos, los modelos usados permiten la inicialización de las tarjetas. Se ha usado su versión en tarjeta y en eToekn (un dispositivo USB que incorpora el equivalente a un chip criptográfico y lector en un mismo dispositivo).
- DNIe. Aunque no era un requisito inicial del proyecto, debido a su similitud con las tarjetas Ceres, se decidió probar los DNIe con la aplicación PKCS#11 desarrollada en el presente proyecto.

Pese a que todas las SmartCards con las que se ha trabajado soportan PKCS#11, hay que decir que hay muchas diferencias en cuanto a cómo interpretan las distintas llamadas PKCS#11 e incluso hay diferencias en los resultados que arrojan esas llamadas. Esto no tiene por qué deberse al hardware en sí de las tarjetas, sino a la capa PKCS#11 implementada en los Cryptoki de cada una de las tarjetas probadas (que como se verá posteriormente muchas de las diferencias de comportamiento radican en la implementación de dicha librería).

Todas las DLLs de las tarjetas dentro de Windows (las Cryptoki) se instalan dentro del directorio C:\Windows\System32\ y desde ahí se deben enlazar en la aplicación para ser utilizadas. Estas DLLs se expresarán en las características de las tarjetas recogidas en el siguiente punto.

En el siguiente apartado se definirán las características técnicas de todos estos modelos utilizados en el proyecto.

3.1 – CARACTERÍSTICAS TÉCNICAS DE LAS SMARTCARDS USADAS

3.1.1 – CERES

MEMORIA	32 Kb
CLAVES ASIMÉTRICAS	RSA
CIFRADO SIMÉTRICO	Triple DES
HASH	SHA1
SOPORTE FIRMA	Sí
PIN (MIN)/PIN (MAX)	8/16
CLAVES RSA (MAX)	2048
INICIALIZACIÓN PERMITIDA	No
CRYPTOKI	PKCSv2GK.DLL

Ilustración 13: Tarjeta Ceres

Tabla 8: Características Ceres

La tarjeta Ceres provee de un software muy básico que permite eliminar certificados, importar perfiles PKCS#12 y eliminar objetos huérfanos (claves sin su par, etc.).

3.1.2 – STARCOS

MEMORIA	8 Kb
CLAVES ASIMÉTRICAS	RSA
CIFRADO SIMÉTRICO	RC4, DES, Triple DES
HASH	MD5, SHA1, SHA256
SOPORTE FIRMA	Sí
PIN (MIN)/PIN (MAX)	4/8
CLAVES RSA (MAX)	1024
INICIALIZACIÓN PERMITIDA	No (Sólo en modelos Developer)
CRYPTOKI	aetpkss1.dll

Ilustración 14: Tarjeta Starcos 2.3 (Blank Card)

Tabla 9: Características Starcos

La tarjeta Starcos dispone de un software algo mejor, que permite visualizar los objetos de la tarjeta (certificados, claves, etc.), eliminarlos e importar PKCS#12 y certificados sueltos. Permite el formateo de la tarjeta (no inicializado, a no ser que sea una versión developer).

3.1.3 – ALADDIN (TARJETA Y ETOKEN)

MEMORIA	72 Kb
CLAVES ASIMÉTRICAS	RSA
CIFRADO SIMÉTRICO	RC4, DES, Triple DES, AES
HASH	MD5, SHA1, SHA256
SOPORTE FIRMA	Sí
PIN (MIN)/PIN (MAX)	Sin límites
CLAVES RSA (MAX)	2048
INICIALIZACIÓN PERMITIDA	Sí
CRYPTOKI	eToken.dll

Ilustración 15: Aladdin eToken

Tabla 10: Características Aladdin

Las tarjetas Aladdin y el eToken Aladdin disponen de un software de gran calidad, que permite visualización y gestión de los objetos presentes en el token, así como el formateo y configuración de gran cantidad de elementos (tratados a nivel PKCS#15) que permiten dar un uso extendido a las tarjetas, como, por ejemplo, establecer la calidad de la clave (número y letras, símbolos, etc.).

3.1.4 – DNIE

CLAVES ASIMÉTRICAS	RSA
HASH	SHA1
SOPORTE FIRMA	Sí
PIN (MIN)/PIN (MAX)	8/16
CLAVES RSA (MAX)	2048
COMENTARIOS	No se permite la gestión de los objetos, sólo el uso de los ya existentes.
CRYPTOKI	UsrPkcs11.dll

Ilustración 16: DNIe

Tabla 11: Características DNIe

El DNIe no aporta software de control PKCS#11, sólo el Cryptoki para que sea usado por aplicaciones de terceros (como sería el presente proyecto). Como curiosidad, mencionar que la DLL del DNIe tiene incorporado un elemento de seguridad por el cual, cuando se va a firmar algo con el DNIe, aparece un mensaje emergente que nos pide una confirmación para realizar la firma.

3.2 – COMPARATIVA DE LAS TARJETAS

En este punto, se va a realizar, por un lado, una comparativa de rendimiento de las tarjetas bajo una batería de pruebas predeterminada (se describirá a continuación). Por otro lado, se harán una serie de comentarios sobre el uso y peculiaridades de cada tarjeta (cómo entienden el estándar PKCS#11, cómo se comportan al utilizarlas, etc.).

Para la comparativa de rendimiento, se han realizado varios tipo de ejecuciones repetidas un número concreto de veces (50 para ser exactos). Todas las pruebas se hicieron con las tarjetas vacías, salvo las que requerían de algún objeto concreto para ser realizadas. Con esto, se pretendió minimizar las latencias debidas a la búsqueda de espacios en el EF correspondiente de PKCS#15. La dinámica de las pruebas consistía en tomar un tiempo de inicio, ejecutar la manipulación del objeto a probar, tomar nuevamente el tiempo, hacer el cálculo de tiempo transcurrido (como tiempo final – tiempo inicial) y eliminar el objeto creado. Esto se realizaba en un bucle 50 veces tomando tiempos distintos para cada iteración. Los tiempos que se muestran fueron los tiempos medios de ejecución de las pruebas y son expresados en milisegundos.

Las pruebas realizadas fueron: la generación de un par de claves RSA de 1024 bits, la generación de un par de 2048 bits, la introducción de un certificado de 2’3 Kb de tamaño dentro de las tarjetas y, finalmente, firmar un fichero de 1 Mb con una clave RSA privada de 1024 bits usando como algoritmo de resumen SHA1.

Para las pruebas, se han probado todos los modelos (salvo el DNIe que no permite manipulación de objetos) y, en el caso de Aladdin, se han probado el modelo de tarjeta y de eToken por separado para comparar el rendimiento de cada uno de ellos.

Del DNIe, puesto que no se han realizado pruebas, se puede comentar que trae incorporados dos certificados (con sus correspondientes claves públicas y privadas), uno enfocado a la firma digital y otro a la autenticación frente a la administración. Las claves que usa son de 2048 bits y no se permite la generación ni incorporación de nuevos elementos dentro del chip. Sus capacidades son las mismas que en las tarjetas Ceres. Por tanto, de haberse realizado pruebas medibles, los resultados hubieran sido muy similares a los de Ceres.

Los datos (en formato tabla y gráfica) obtenidos, tras las 50 ejecuciones en cada tarjeta, fueron los siguientes:

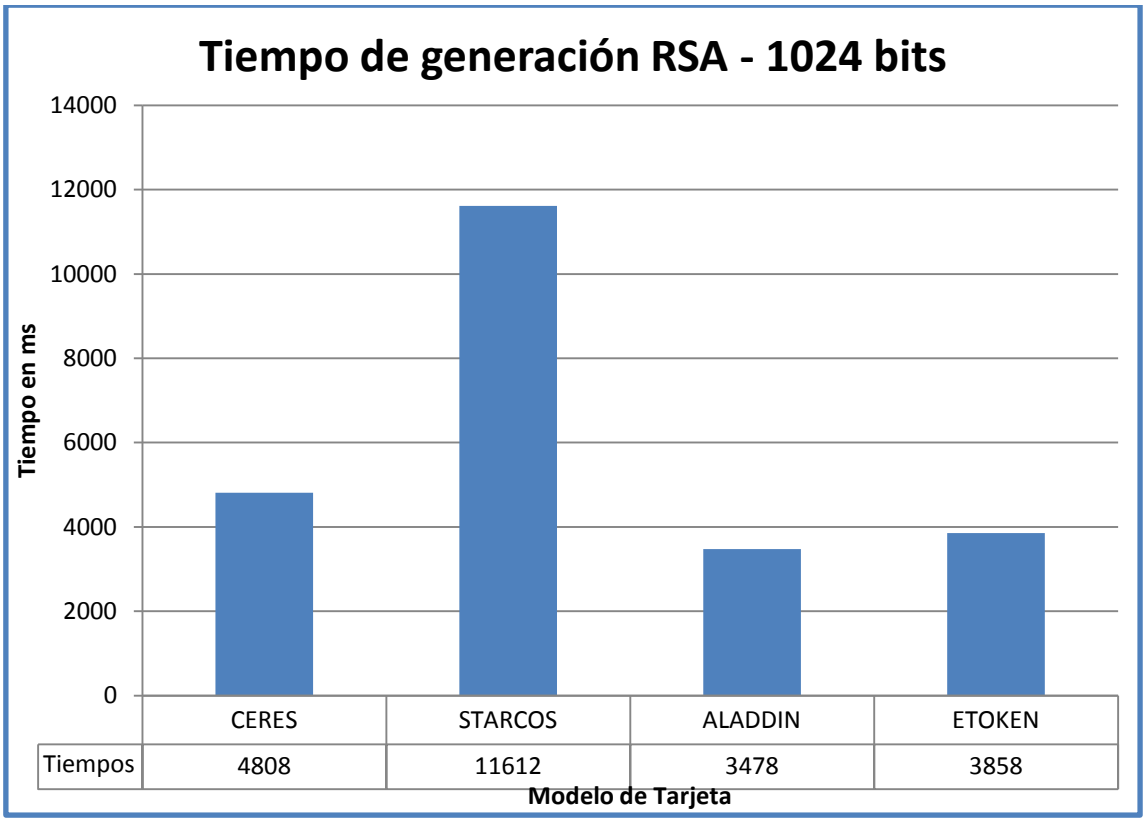

Ilustración 17: Tiempo de generación RSA - 1024 bits

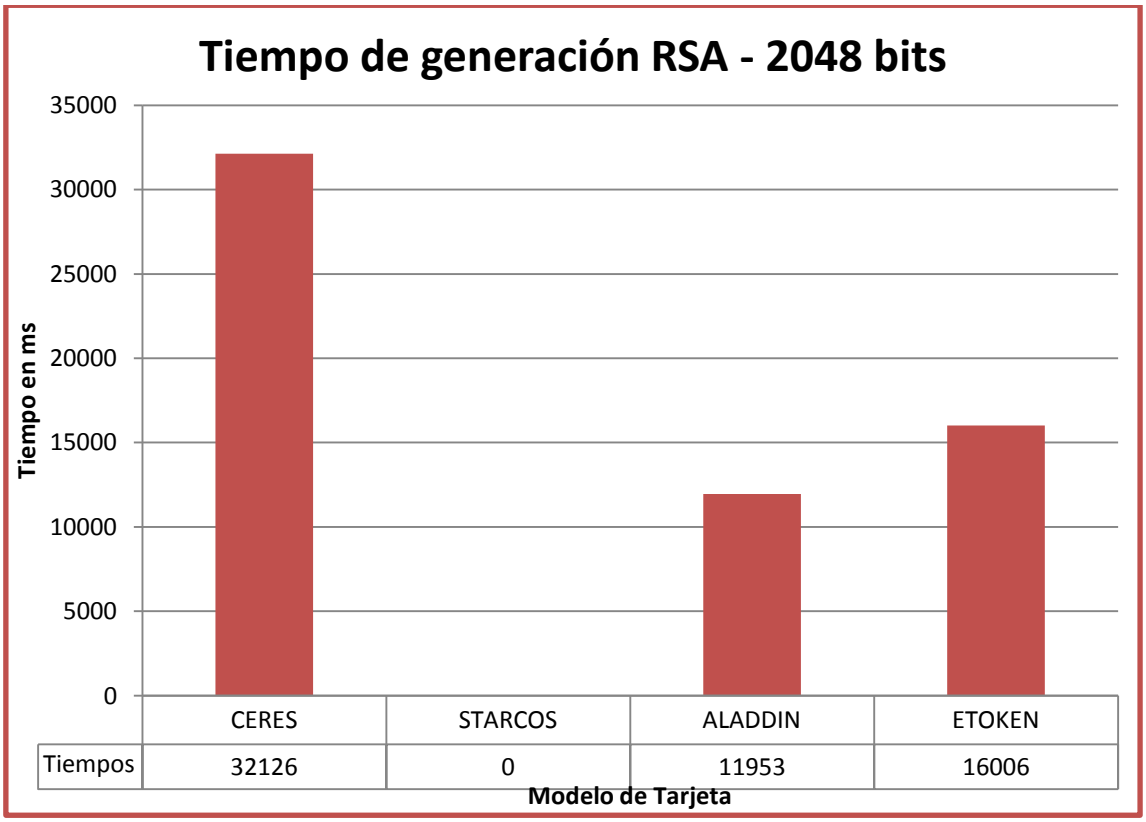

Ilustración 18: Tiempo de generación RSA - 2048 bits

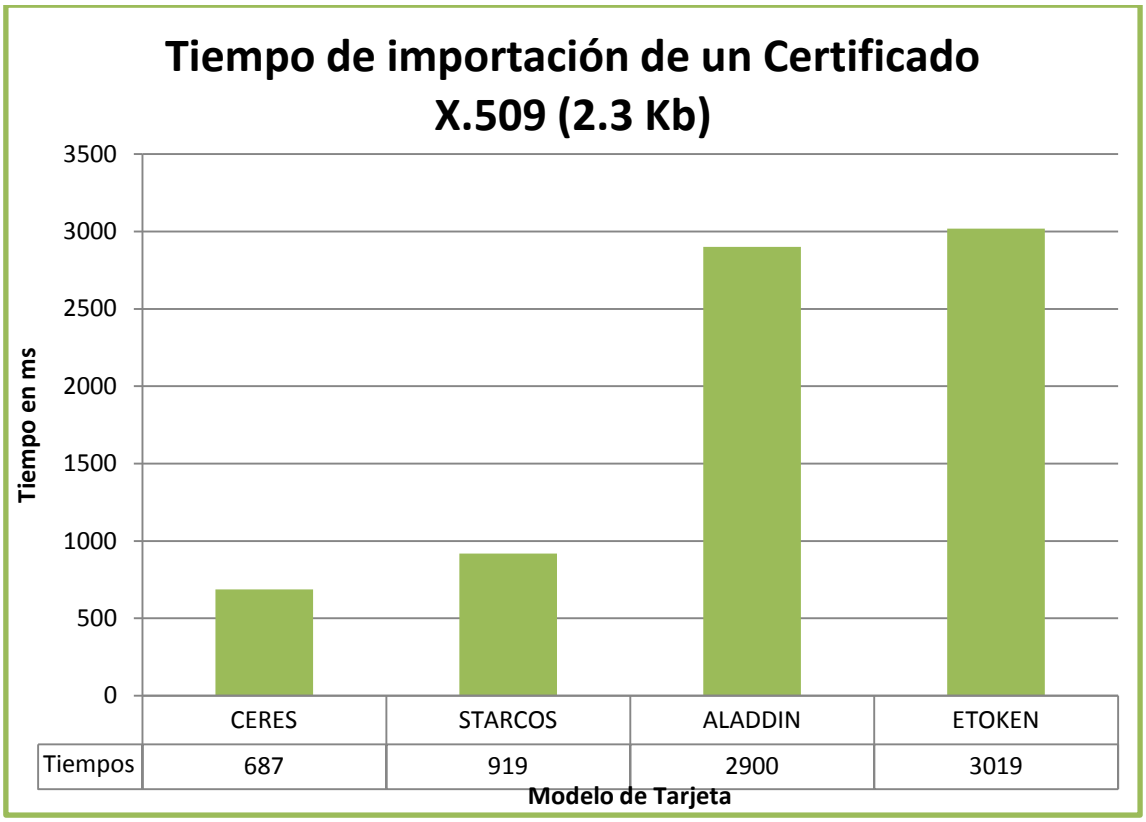

Ilustración 19: Tiempo de importación de un Certificado X.509 (2.3 Kb)

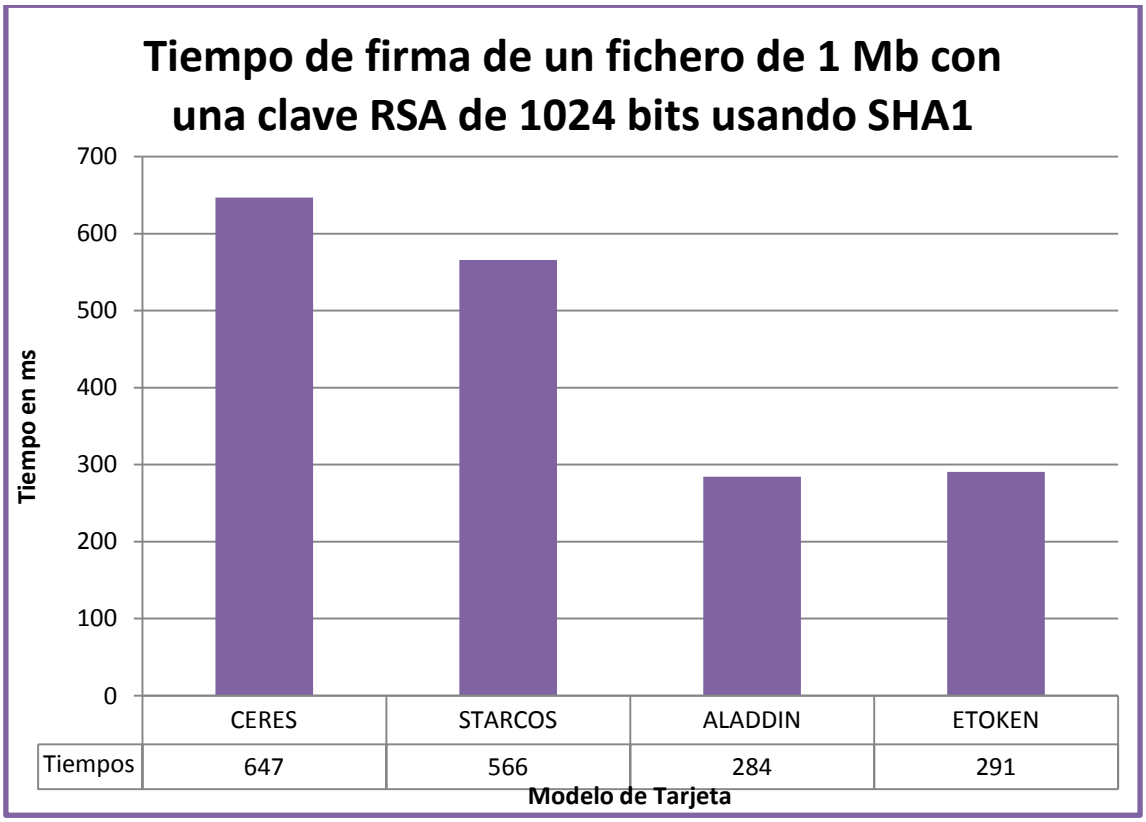

Ilustración 20: Tiempo de firma de un fichero de 1 Mb con una clave RSA de 1024 bits usando SHA1

Como se puede ver, a simple vista, los mejores tiempos, en general, fueron obtenidos por las tarjetas Aladdin (mejores en el caso de la tarjeta que en el eToken). Las Ceres y Starcos son entre sí más eficientes en algunas pruebas y menos en otras, salvo que Starcos no soporta la generación de claves RSA de 2048 bits.

Aladdin tiene sólo un inconveniente (que para el uso cotidiano de las tarjetas no es demasiado significativo), es tremendamente ineficiente a la hora de importar/eliminar objetos, pero no al usarlos (donde fue la más eficiente).

Si en la comparativa incluimos las características técnicas, la balanza se decanta más aún del lado de las Aladdin que son, con diferencia, las que más funcionalidad hardware soportan. En este caso, la Tarjetas Ceres son muy inferiores al resto, ya que permiten muy pocas operaciones (que se limitan a la creación de claves y al firmado).

El gran inconveniente de las tarjetas Ceres es que la DLL PKCS#11 que tienen se comporta francamente mal (hacen una interpretación muy libre del estándar) a la hora de gestionar las tarjetas (no tanto a la hora de hacer un uso habitual de las mismas). Entre otras cosas, la DLL no permite usar claves recién creadas sin "reiniciar" toda la aplicación que invoca a la DLL (hasta que no se libera la DLL no puede hacer uso correcto de las tarjetas).

La librería de Ceres tiene otros comportamientos no deseados (que no son malos en sí, pero al no seguir el estándar hacen que produzca incompatibilidades) que serían:

- Al importar una clave privada de forma automática, genera la pública.
- Al eliminar una clave privada, igualmente, elimina la pública.
- El Secure Officer y el Usuario Normal comparte PIN, de modo que no se puede desbloquear un PIN bloqueado mediante el estándar PKVS#11 (habría que recurrir a mecanismos no estandarizados).

En cuanto a las plantillas utilizadas para la gestión de los objetos, todas las tarjetas se comportaron de forma bastante homogénea, salvo las Aladdin que, en general, se mostraron incompatibles ante la intrusión de plantillas sobrecargadas (con mucha información no necesaria).

Para concluir, se puede decir que tanto en hardware, como en uso, como por calidad de la librería PKCS#11, las mejores tarjetas probadas, en tanto a lo que se refiere a PKCS#11, como lo que se refiere a la integración general en el sistema, son las tarjetas Aladdin (concretamente es mejor la versión SmartCard que la eToken).

4 – ANÁLISIS DEL PROYECTO

En esta parte del proyecto, se van a exponer todos los procedimientos realizados y todos los elementos tenidos en cuenta para entender exactamente la magnitud de lo que se intenta resolver.

Se va a definir lo que se pretende que debe hacer la aplicación de gestión de PKCS#11, cómo debe hacerlo, dónde se va a desplegar (o utilizar) dicha aplicación y el tipo de usuario que va a utilizar la aplicación.

Para evitar que el análisis sea demasiado extenso (la presente memoria no puede tener cientos y cientos de páginas porque su objetivo es sintetizar no detallar de forma milimétrica) se van a tratar seis apartados, de los cuales sólo el último se hará con relativa extensión. El primero va a ser describir las características generales que tiene el sistema. El segundo las restricciones que tiene. Luego se describirá el entorno operacional donde deberá ejecutarse el sistema. A continuación, se expondrán las características de los usuarios. Después se dará un enfoque de cómo debe ser la solución y finalmente un diagrama de casos de uso con sus descripciones textuales para poder entender realmente para qué va a servir el sistema.

4.1 – CAPACIDADES DEL SISTEMA

La orientación de la aplicación que se describe en esta memoria es la gestión de tarjetas SmartCards mediante el uso del estándar PKCS#11. Por tanto, todas las capacidades del sistema (que podrían mapearse en un análisis extenso con requisitos de capacidad) irán enfocadas al uso de dichas tarjetas.

Además de la gestión interna de las tarjetas, también deben gestionarse todas aquellas operaciones necesarias para el correcto funcionamiento de las tarjetas, como son la gestión de certificados y elementos que van a ser introducidos del exterior (en forma de ficheros desde un ordenador).

Estas capacidades del Sistema serán vistas de forma detallada (e incluso capacidades a más bajo nivel) dentro del apartado de casos de uso. Aquí se va a hacer una breve introducción a qué va a poder hacer el usuario dentro de la aplicación.

Las capacidades se pueden dividir en varios bloques según su finalidad:

- Capacidades de acceso a la tarjeta.
  - Login: El usuario podrá acceder a la tarjeta.
  - Logout: El usuario podrá abandonar su sesión en la tarjeta.
  - Desbloquear PIN: El usuario podrá rehabilitar PINes bloqueados.
  - Cambio de PIN: El usuario podrá modificar el PIN de la tarjeta.
- Capacidades de Gestión de los objetos de la tarjeta.
  - Obtener listado de objetos: el usuario podrá obtener una lista con los objetos presentes en la tarjeta, pudiendo ver sus principales características. En aquellos de mayor relevancia, podrá verlos de forma detallada (en certificados y claves públicas).
  - Modificación de los objetos: El usuario podrá modificar ciertas características (atributos) de los objetos de la tarjeta (los IDs y las etiquetas).
  - Borrado de elementos: El usuario podrá borrar cualquier elemento de la tarjeta.
- Capacidades informativas de la tarjeta:
  - Información de la tarjeta: el usuario podrá obtener información de la tarjeta, así como información relacionada con los procesos hardware soportados por la tarjeta y, además, ver información del lector usado para leer la tarjeta.
  - Generación de números aleatorios: El usuario podrá generar números pseudoaleatorios dentro de la tarjeta.
- Capacidades de gestión de claves y certificados:

- Generación de claves: el usuario podrá generar pares de claves RSA dentro de la tarjeta.
- Crear CSR: el usuario podrá generar peticiones PKCS#10 a partir de las claves generadas en la tarjeta (también podrían haber sido importadas). Los CSR podrán ser para propósito general (como firma digital) o para hacer logon en sistemas Windows.
- Importar certificados: el usuario podrá importar certificados X.509 dentro de la tarjeta (para asociarlos, si quiere, con pares de claves).
- Exportar certificados: el usuario podrá extraer de la tarjeta cualquier certificado X.509 que en ella haya.
- Importar PKCS#12: el usuario podrá importar perfiles PKCS#12 desde ficheros, para así introducir clave pública, privada y certificado de una persona o entidad.

Estas son todas las capacidades funcionales que deberá tener la aplicación. En los casos de uso se verán de forma detallada exponiendo las diferentes situaciones de ejecución, los efectos, etc. Esto se hará así (como ay se menciono) para evitar la redundancia lo más posible y hacer esta memoria más ligera y menos sobrecargada.

4.2 – RESTRICCIONES DEL SISTEMA

En este apartado, se van a tratar las restricciones generales del sistema (que se podrían mapear con los requisitos de restricción de un análisis realizado de forma más extensa).

Se van a tratar dos tipos de restricciones básicas en este proyecto: las restricciones de formato (codificación) y las restricciones de seguridad. En principio, no se plantean restricciones al funcionamiento en sí de la aplicación.

Hay que decir que muchas restricciones de funcionamiento vienen impuestas por el propio hardware de las tarjetas y no en sí por capacidades software desarrolladas en la presente aplicación o implementadas en los distintos Cryptoki de PKCS#11. Estas serán analizadas en el siguiente punto.

Las restricciones de formato serían las siguientes:

- Todos los fichero de certificados (X.509) que vayan a ser importados dentro de la tarjeta deberán tener una codificación PEM (Base64).
- Los ficheros de Identidad digital (PFX) con perfiles PKCS#12 deberán tener una codificación DER.
- Las firmas digitales se almacenarán en ficheros (siempre). Estos ficheros tendrán una codificación binaria (traducción directa a ASCII).

En cuanto a las restricciones de seguridad:

- Los PINes deben ser suministrados a la tarjeta de forma segura (al menos de modo que no sean visibles). El resto de la seguridad dependería del sistema operativo y las librerías Cryptoki.

4.2.3 – RESTRICCIONES HARDWARE DE LAS TARJETAS

Todas las SmartCards, y más concretamente las que trabajan con el estándar PKCS#11, tienen una serie de restricciones de seguridad generales que son invariantes y que deben ser tendías en cuenta a la hora de poder analizar el uso de las tarjetas.

A continuación, se enumeran las restricciones más significativas (las que tienen que ver con la seguridad) que tiene el hardware de las SmartCards:

- Las tarjetas requieren siempre de un PIN para poder acceder como usuario (ya sea Oficial de Seguridad o usuario Normal).

- El número de intentos de acceso mediante PIN erróneo está limitado. Suele ser de 3 intentos fallidos (es personalizable en el formateo de la tarjeta en aquellas que se permite). Pasados esos intentos el PIN se bloquea haciendo imposible acceder a la tarjeta (hasta el desbloqueo).
- Ya sea mediante PKCS#11 o mediante mecanismos propietarios no públicos la clave de acceso (o PIN) y la clave usada para desbloquear un PIN son diferentes (pueden coincidir sus caracteres, pero serán elementos diferentes dentro de la tarjeta).
- Las claves privadas siempre tienen el atributo `CKA_PRIVATE` con valor `CK_TRUE`. Esto supone que las claves privadas sólo podrán ser usadas mediante la inserción del PIN.
- Las claves privadas no serán visibles (no se podrá tener acceso al valor de sus tributos de clave RSA). Tendrán el atributo `CKA_SENSITIVE` con valor siempre `CK_TRUE`.
- Las claves privadas no serán modificables. El atributo `CKA_MODIFICABLE` será siempre `CK_FALSE`.
- En las claves privadas, como ya se ha dicho, los atributos de clave privada RSA no son visibles. Por tanto, tampoco podrán ser exportados. Esto supone que ninguna clave privada puede ser extraída de la tarjeta al exterior.
- El número máximo de bits del módulo de las claves está limitado por la tarjeta (en algunas a 1024 y en otras a 2048).
- Todas las tarjetas tienen dos tipos de usuarios: el Usuario Normal y el Oficial de Seguridad.
- Todas las codificaciones internas de los objetos, tales como claves o certificados, serán siempre DER.
- Los tipos de objetos están limitados a los descritos en el apartado de PKCS#11<<, de modo que no se podrán crear tipos nuevos.

4.3 – USUARIOS

En principio, la aplicación está enfocada a que un único tipo de usuario estándar la utilice para gestionar SmartCards. Por tanto, desde esa perspectiva no habría realmente diferentes roles dentro de la aplicación.

Como ya se ha visto, las SmartCards tienen dos tipos de usuarios: el Usuario Normal y el Oficial de Seguridad. Al haber dos tipos de usuarios que pueden hacer logan en la tarjeta, se puede entender como que realmente sí que hay dos tipos de roles (al menos) en la aplicación: uno que se autentica como Usuario Normal y otro como Oficial de Seguridad. Ambos podrán ser el mismo. Esto dependería del uso de las tarjetas y del uso de la aplicación, pero esto queda fuera del alcance del proyecto.

Existe también la funcionalidad básica ejecutable en las tarjetas sin haberse autenticado, relacionada toda ella con los objetos públicos. Esto, desde cierto punto de vista, podría entenderse como un rol más dentro de la aplicación.

Todas las capacidades y restricciones de los diferentes usuarios a nivel PKCS#11 (que es quien hace la verdadera distinción entre roles) están definidas en su correspondiente apartado de usuarios del punto relativo a PKCS#11.

4.4 – ENFOQUE DE LA SOLUCIÓN

El principal objetivo de este proyecto era crear una capa que permitiera interactuar con las diferentes SmartCards por medio de PKCS#11 (o mejor dicho, de sus librerías Cryptoki). La idea es que esta capa intermedia de funcionalidad pudiera servir como una especie de API utilizada en cualquier tipo de aplicación para dotarla de capacidad de interactuar con las tarjetas.

El entorno de ejecución, en principio, es Windows, pero el código tendría que ser lo más genérico posible para poder ser portado a sistemas Unix sin problemas (sólo haría falta recompilar, no reescribir).

La necesidad de tener que utilizar DLLs e invocar su funcionalidad a bajo nivel hizo que opciones, como programación Java, quedasen rápidamente descartadas. Java dispone de funcionalidad para interactuar con DLLs que implementen PKCS#11 y poder utilizar tarjetas, pero la interacción se produce a través de su propia API, de modo que no se puede tener acceso directo a la tarjeta imposibilitando un control total de la misma.

El tener que escribir un código portable a Unix descartaba opciones del ámbito .NET como podrían haber sido C#.

Se podría plantear el uso de ANSI C o ANSI C++ para que el código implementado fuera fácilmente portable a otros sistemas operativos que no fuera Windows. Debido a que la solución se basa en el uso de un estándar muy estático, que no va a cambiar ni a corto ni a medio plazo y que la capa de aplicación se va a generar con toda la funcionalidad necesaria, se decidió usar ANSI C en lugar de C++ debido a la facilidad de implementación que tiene el ahorrarse un diseño orientado a objetos de buen nivel.

Todos los proyectos que interactúan con PKCS#11 están implementados en ANSI C. De todos ellos se enumerarán algunos: las DLLs de las tarjetas (todos los Cryptoki están implementados en C), el proyecto OpenSC, que trata de crear una capa de interacción con las tarjetas, también está escrito en C y finalmente Mozilla Firefox, toda la parte de gestión de tarjetas (para el uso de navegación autenticada por internet) está también escrito en C.

Por todo lo dicho, la elección del lenguaje de desarrollo de la aplicación fue C. Que el lenguaje fuera C tiene una serie de inconvenientes, que se resumen en la mantenibilidad de la aplicación. Para evitar que esto se convierta en un problema, debe utilizarse C del modo más modular y ordenado posible, de modo que permita la adicción de nuevas funcionalidades o modificación de las existentes. En el caso de que el estándar requiriera de más información lógica (como tipos de objetos, etc.) o

hubiera sido un entorno más cambiante, habría sido imprescindible el uso de C++, pero un diseño correcto hecho en C permite añadir o modificar funcionalidad de forma rápida y segura.

4.4.1 – TIPO DE INTERFAZ

Debido a que lo esencial del proyecto es la capa de interacción con PKCS#11 y, en menor medida, la gestión de ficheros con certificados, PKCS#12, etc., no era necesario la implementación de una interfaz de usuario. Además, se planteaba el problema de programar una interfaz de usuario que fuera tan portable como el código ANSI C y eso es una tarea muy compleja (sin solución realmente viable). Por tanto, para este proyecto no hay presente una interfaz de usuario, no al menos en el sentido amplio de la palabra (no hay interfaz gráfica de usuario). No obstante, para poder interactuar con la capa de PKCS#11 y también con la de gestión de certificados, sí que se requiere de algún mecanismo de interacción. Para ello, se planeta dos soluciones: una ejecución basada en comandos por consola o una interfaz textual de consola.

El formato de ejecución, basado en comandos (similar al que usa OpenSC), es muy incómodo y poco práctico para una interacción dinámica (sólo sirve para que sea ejecutado como comando por terceras aplicaciones). Esto se debe a que los objetos tienen asicados identificadores, que son la forma de referenciarlos. Esto supone que para interactuar con algún objeto (por ejemplo para firmar algo) primero hay que ejecutar el comando que nos lista los objetos (para obtener su identificador) y con su identificador ejecutar un nuevo comando que use dicho objeto para, por ejemplo, realizar la firma de un fichero de datos.

Así que la solución más viable resulta ser (aunque ciertamente algo incómoda de usar) una interfaz de consola basada en texto.

4.5 – CASOS DE USO

En esta sección de casos de uso, se van a presentar tres diagramas: en el primero, se mostrarán las funcionalidades que realiza el usuario que no está autenticado en la tarjeta. En el segundo, se verán las funcionalidades realizadas por el Usuario Normal y, en el último, las del Oficial de Seguridad. Se han planteado las funcionalidades y la distribución de roles en función de cómo trabajan realmente las tarjetas, no de cómo dice el estándar PKCS#11 que son las cosas (véase el apartado PKCS#11 punto 2.6).

Después de los diagramas, se realizará una descripción textual descriptiva de cada uno de los casos de uso, con la que se entenderá mejor la finalidad de dicho caso.

Hay tres roles definidos: el Usuario sin sesión que se ha denominado en estos casos de uso, que no es más que el rol en PKCS#11 de aquel usuario que no ha hecho login en la tarjeta. Este usuario puede acceder a toda la información pública de la tarjeta, e incluso usar todos los objetos públicos. El Usuario Normal, es el que puede hacer la mayor parte de la gestión de objetos en la tarjeta, debido a que tiene acceso tanto a objetos públicos como a los privados (requiere del PIN). Finalmente, el Oficial de Seguridad puede interactuar sobre los objetos públicos (no sobre los privados) y su principal función es la de poder desbloquear los PINes bloqueados. Además el oficial de Seguridad puede realizar todas las opciones de gestión de objetos, pero solamente sobre los objetos públicos.

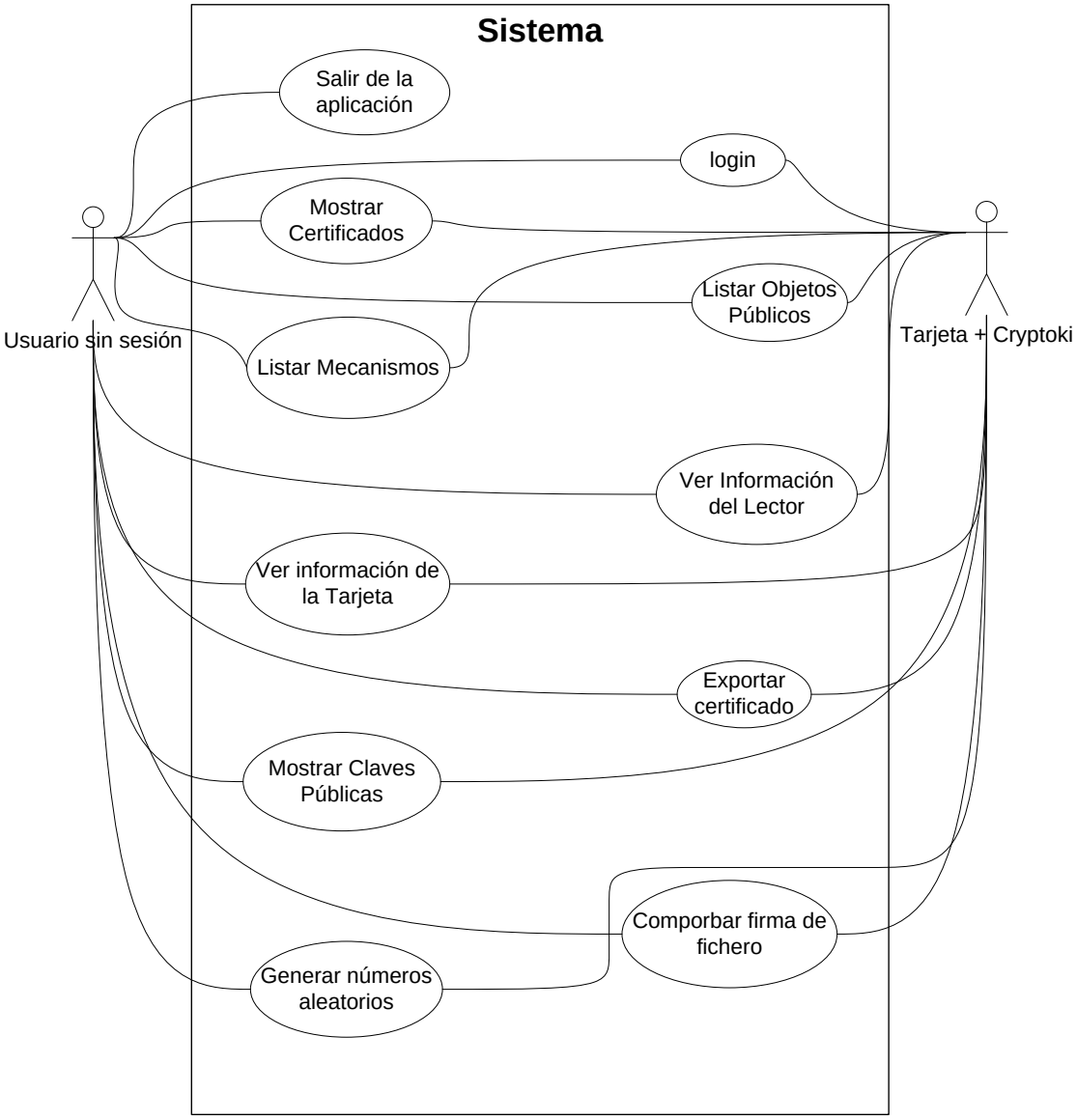

Ilustración 21: Casos de uso del Usuario sin sesión

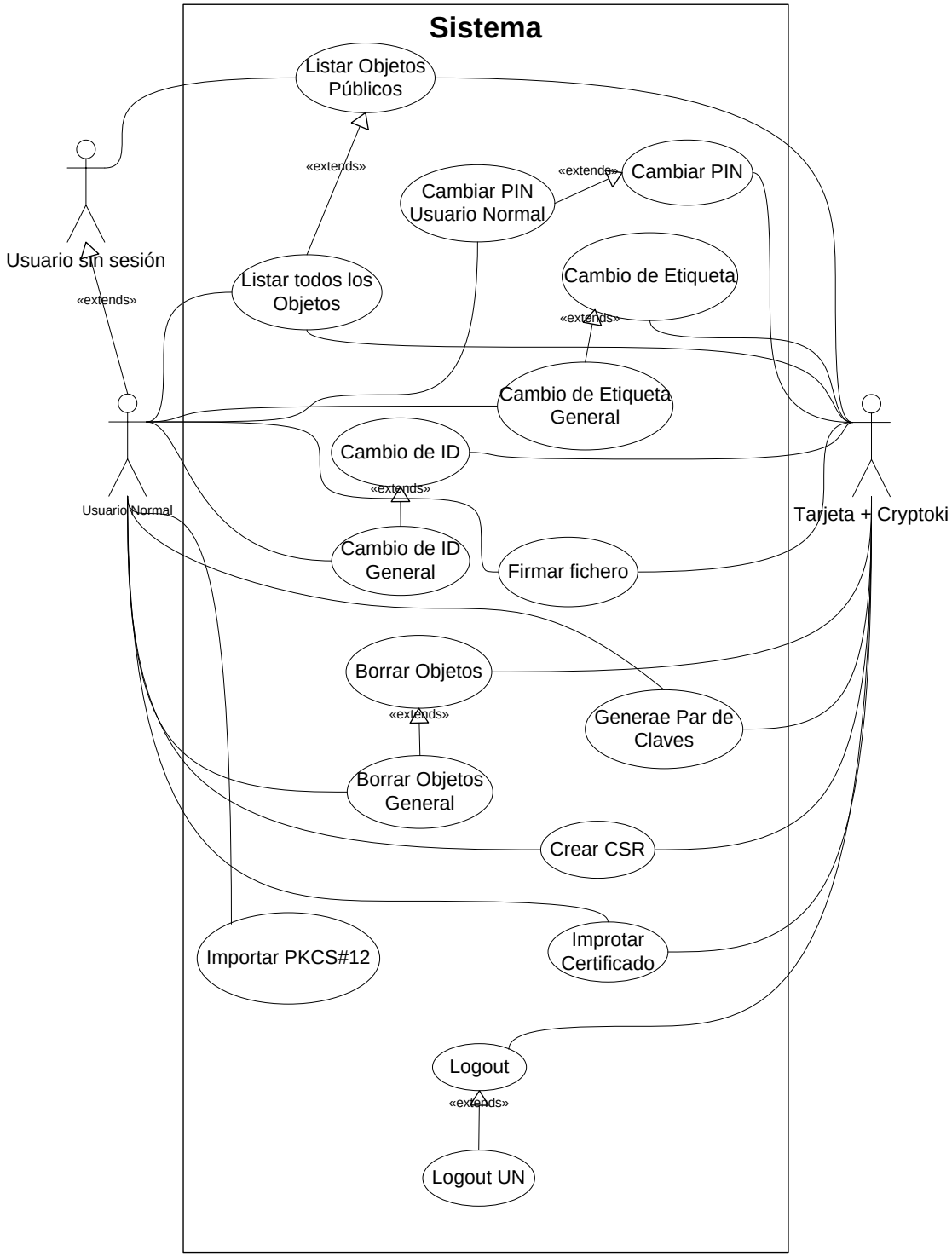

Ilustración 22: Casos de uso del Usuario Normal

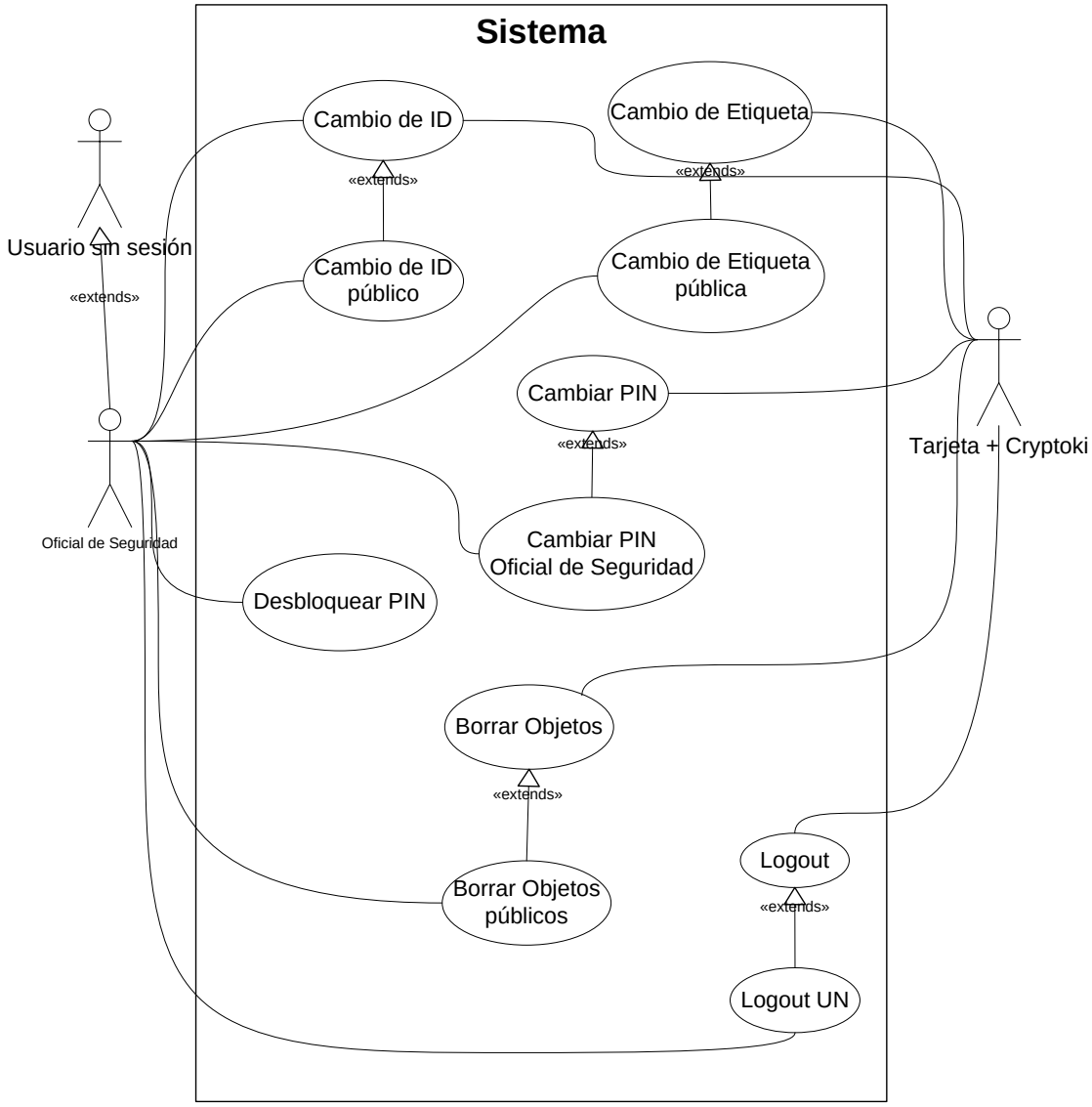

Ilustración 23: Casos de uso del Oficial de seguridad

En los diagramas, hay una serie de funcionalidades (Cambio de ID, Cambio de Etiqueta y Borrar Objetos) que son realizadas en colaboración con la tarjeta, pero que ningún otro actor colabora. Sin embargo, estas acciones son extendidas por otras dos (cada una de ellas tiene una extensión a General y a público). Estas acciones son realizadas por El Usuario Normal y por el Oficial de Seguridad (las generales por el usuario Normal y las públicas por el Oficial de Seguridad). Esto hace referencia a que todas las acciones están en colaboración con la tarjeta; pero, para reducir las líneas en el diagrama; sólo se han representados las que enlazan el caso de uso de mayor nivel.

Todas esas acciones son realmente las mismas, pero se diferencian en pequeños matices. Borrar General, por ejemplo, hace referencia al borrado de objetos en la tarjeta (sin distinción) es una acción abstracta, por lo que ningún actor activo del

sistema la ejecuta. Las acciones concretas serían Borrado General, realizado por el Usuario Normal que representa el borrado de cualquier objeto de la tarjeta. Por el contrario, el Borrar Objetos Públicos representa la misma acción, aunque nunca podría realizarse sobre objetos privados (simplemente porque no son mostrados por la tarjeta).

4.5.1 – DESCRIPCIÓN TEXTUAL DE LOS CASOS DE USO

Para realizar las descripciones de los casos de uso, se van a utilizar tablas con los siguientes atributos:

- **Nombre**: Es el nombre genérico del caso de uso (la acción a realizar).
- **Actores**: Los actores que intervienen en la realización del caso de uso (quién realiza la acción).
- **Descripción**: Una breve reseña de lo que hace el caso de uso (qué se hace).
- **Precondiciones**: Descripción del estado del sistema para realizar la acción (en qué condiciones se puede realizar la acción).
- **Postcondiciones**: Estado en el que queda el sistema tras realizar la acción (efectos de la acción).
- **Escenario Básico**: Ejecución habitual satisfactoria de la acción (pasos que se dan en la ejecución correcta de la acción).
- **Escenario Alternativo**: Ejecución errónea de la acción (pasos a realizar por una acción que no obtiene un resultado satisfactorio por alguna causa concreta).

No se ha incluido un campo típico de las descripciones textuales, el código de la acción. Esto se debe a que no hay requisitos de usuario (se han especificado directamente con capacidades y en estos casos de uso) por lo que no va a haber trazabilidad de requisitos y no es necesario incluir ese código.

Nombre	Salir de la aplicación.
Actores	Usuario sin sesión.
Descripción	Una vez dentro del sistema se tiene que poder salir de la aplicación.
Precondiciones	Haber abierto la aplicación.
Postcondiciones	El sistema se cierra completamente.
Escenario Básico	No aplica.
Escenario Alternativo	No aplica.

Tabla 12: Caso de Uso "Salir de la aplicación"

Nombre	Login.
Actores	Usuario sin sesión y Tarjeta+Cryptoki.
Descripción	Por medio del login y suministrando el PIN se accede a las funcionalidades restringidas de la tarjeta, ya sean las del Usuario Normal (acceso a objetos privados, etc.) como las del Oficial de Seguridad (desbloqueos de PIN, etc.).
Precondiciones	No estar previamente autenticado en la tarjeta.
Postcondiciones	El usuario pasa a estar autenticado como Usuario Normal u Oficial de Seguridad (dependiendo de la elección).
Escenario Básico	• Se selecciona el tipo de usuario. • Se suministra el PIN correcto. • Se tiene acceso a la funcionalidad extendida (autenticado en el sistema).
Escenario Alternativo	• Se selecciona el tipo de usuario. • Se suministra un PIN incorrecto. • Se notifica que el PIN no es válido.

Tabla 13: Caso de Uso "Login"

Nombre	Mostrar Certificados.
Actores	Usuario sin sesión y Tarjeta+Cryptoki.
Descripción	Se visualizará una estructura decodificada de todos los campos de un certificado X.509 contenido en la tarjeta.
Precondiciones	Que haya al menos un certificado en la tarjeta.
Postcondiciones	Se verán todos los campos de certificado X.509 con nombres descriptivos de sus atributos y sus correspondientes valores.
Escenario Básico	• Se selecciona el Certificado correspondiente. • Se obtiene una decodificación legible del mismo.
Escenario Alternativo	• No hay ningún certificado dentro de la tarjeta.

Tabla 14: Caso de Uso "Mostrar Certificados"

Nombre	Listar objetos Públicos.
Actores	Usuario sin sesión y Tarjeta+Cryptoki.
Descripción	El usuario obtiene un listado de todos los objetos públicos de la tarjeta obteniendo su etiqueta, su ID, el tipo de datos y en Número de Serie en caso de ser un certificado. Por defecto, todos los objetos salvo las claves privadas son públicos.
Precondiciones	No aplica.
Postcondiciones	Se obtiene un listado de los objetos públicos de la tarjeta.
Escenario Básico	• Se obtiene un listado con todos los objetos públicos de la tarjeta.
Escenario Alternativo	• La tarjeta está vacía o no tiene objetos públicos.

Tabla 15: Caso de Uso "Listar Objetos Públicos"

Nombre	Listar mecanismos.

Actores	Usuario sin sesión y Tarjeta+Cryptoki.
Descripción	El usuario puede visualizar una lista de los mecanismos soportados por la tarjeta y el Cryptoki. Los mecanismos generalmente hacen referencia a las capacidades hardware de la tarjeta.
Precondiciones	No aplica.
Postcondiciones	Se obtiene un listado con todas las funcionalidades hardware y del Crytpoki que pueden ser ejecutadas, tales como firma basada en SHA1, creación de claves, etc.
Escenario Básico	• Se obtiene un listado de los mecanismos de la tarjeta.
Escenario Alternativo	No aplica.

Tabla 16: Casos de Uso "Listar Mecanismos"

Nombre	Ver Información del Lector.
Actores	Usuario sin sesión y Tarjeta+Cryptoki.
Descripción	Se obtiene información referente al lector que se está usando para leer la tarjeta. La información que se obtiene es: Nombre del Lector, Versión del Firmware, Versión de Hardware e ID del fabricante.
Precondiciones	No aplica.
Postcondiciones	Se obtiene información del lector de tarjetas en uso.
Escenario Básico	• Se obtiene un listado de las características del lector.
Escenario Alternativo	No aplica.

Tabla 17: Caso de Uso "Ver Información del Lector"

Nombre	Ver Información de la Tarjeta.
Actores	Usuario sin sesión y Tarjeta+Cryptoki.
Descripción	Se obtiene información referente a la tarjeta que se está usando. La información que se obtiene es: Nombre del Lector, Versión del Firmware, Versión de Hardware e ID del fabricante.
Precondiciones	No aplica.
Postcondiciones	Se obtiene información de la tarjeta en uso.
Escenario Básico	• Se obtiene un listado de las características de la tarjeta.
Escenario Alternativo	No aplica.

Tabla 18: Caso de Uso "Ver información de la Tarjeta"

Nombre	Mostrar Clave Pública
Actores	Usuario sin sesión y Tarjeta+Cryptoki.
Descripción	Se obtiene información de la clave pública. Concretamente se obtendrán el módulo y el exponente en formato hexadecimal.
Precondiciones	Que exista al menos una clave pública en la tarjeta.
Postcondiciones	Se ven los datos relativos a la clave pública seleccionada.
Escenario Básico	• Se obtienen el módulo y el exponente de la clave pública.

Escenario Alternativo	• No hay claves públicas en la tarjeta.

Tabla 19: Caso de Uso "Mostrar Clave Pública"

Nombre	Exportar Certificado.
Actores	Usuario sin sesión y Tarjeta+Cryptoki.
Descripción	Se generará un fichero en el ordenador que ejecuta la aplicación que contendrá el certificado exportado de la tarjeta en codificación PEM.
Precondiciones	Que exista al menso un certificado público en la tarjeta.
Postcondiciones	Se genera un fichero con la codificación PEM del certificado. Este fichero podrá ser utilizado en el futuro por el sistema operativo o por la propia aplicación.
Escenario Básico	• Se selecciona el certificado a exportar. • Se genera selecciona el nombre de fichero y ruta. • Se genera el fichero con la codificación PEM del certificado.
Escenario Alternativo	• No existe la ruta expresada para el fichero de salida.

Tabla 20: Caso de Uso "Exportar Certificado"

Nombre	Generar Números Aleatorios
Actores	Usuario sin sesión y Tarjeta+Cryptoki.
Descripción	El usuario podrá generar números pseudoaleatoios, que serán mostrados por pantalla en formato binario (codificación de 1s y 0s). Se podrán generar número de entre 1 y 64 bytes (lo que es lo mismo de entro 8 y 512 bits).
Precondiciones	No aplica.
Postcondiciones	Se genera el número en formato binario (por tanto agnóstico de la codificación, ya sea real, completo a 1, complemento a 2, etc.).
Escenario Básico	• Se selecciona el número de bytes del número 8de 1 a 64). • Se obtiene la codificación binaria (de entre 8 y 512 bits).
Escenario Alternativo	No aplica.

Tabla 21: Caso de Uso "Generar Números Aleatorios"

Nombre	Comprobar Firma de Fichero
Actores	Usuario sin sesión y Tarjeta+Cryptoki.
Descripción	El usuario podrá comprobar la firma de un fichero, que estará contendía a su vez en otro fichero diferente con codificación ASCII. Se obtendrá una validación de la firma para una clave pública y mecanismo determinado si es válida o por el contrario signature invalid si no lo es.
Precondiciones	Que haya una clave pública en la tarjeta. Que exista un fichero para comprobar. Que exista un fichero con la firma.
Postcondiciones	Se obtendrá una validación o un rechazo de la firma digital del fichero.
Escenario Básico	• Se selecciona el fichero del cual se desea comprobar la firma

	• Se selecciona la clave pública dentro de la tarjeta • Se selecciona el mecanismo de comprobación de la firma • Se selecciona el fichero que contiene la firma digital • El programa valida o rechaza la firma digital

Escenario Alternativo	• Se selecciona el fichero para comprobar la firma.
	  ○ El fichero o la ruta no existen.
	• Se selecciona el fichero que contiene la firma
	  ○ El fichero tiene menso bytes de los necesarios para comprobar la firma y produce un error.

Tabla 22: Caso de Uso "Comprobar Firma de Fichero"

Nombre	Crear CSR
Actores	Usuario Normal y Tarjeta+Cryptoki.
Descripción	A partir de una clave pública, el usuario puede crear una petición de certificado PKCS#10. Esta podrá ser para propósito general (firma digital) o para logon en sistemas Windows. Se suministraran datos referentes al certificado futuro y se utilizará la clave privada asociada a la pública usada en el CSR para firmadlo.
Precondiciones	Estas autenticado en el sistema como Usuario Normal. Tener una clave pública y su privada asociada.
Postcondiciones	Se generará un fichero en formato PEM con la petición PKCS#10.

Escenario Básico	• Se selecciona la clave pública.
	• Se selecciona el tipo de certificado (logon o firma)
	• Si se selecciono firma:
	  ○ Se suministra el país.
	  ○ Se suministra la provincia.
	  ○ Se suministra la localidad.
	  ○ Se suministra el nombre de la organización.
	  ○ Se suministra el departamento de la organización.
	  ○ Se suministra un nombre.
	  ○ Se suministra un email.
	• Si se seleccionó Logon:
	  ○ Se suministra el grupo de usuarios.
	  ○ Se suministra el usuario para logon.
	  ○ Se suministra la extensión del dominio.
	  ○ Se suministra el nombre del dominio.
	• Se selecciona la clave privada para firmar la petición.
	• Se Introduce la ruta y el nombre del fichero que contendrá la petición.

Escenario Alternativo	• No hay clave pública para generar la partición o no hay una privada para firmarla. • Tras los datos, se selecciona una ruta o fichero incorrectos.

Tabla 23: Caso de Uso "Crear CSR"

Nombre	Listar todos los Objetos
Actores	Usuario Normal y Tarjeta+Cryptoki.
Descripción	Esta funcionalidad permite visualizar todos los objetos contenidos dentro de la tarjeta (incluidos los objetos privados). La gran diferencia sobre Visualizar Objetos, es que permite ver las claves privadas.
Precondiciones	Estar autenticado en el sistema como Usuario Normal.
Postcondiciones	Se obtiene un listado con los objetos de la tarjeta.
Escenario Básico	• El Usuario Normal se autentica en el sistema. • Se obtiene una lista de los objetos de la misma.
Escenario Alternativo	No aplica.

Tabla 24: Caso de Uso "Listar todos los Objetos"

Nombre	Cambiar PIN Usuario Normal
Actores	Usuario Normal y Tarjeta+Cryptoki.
Descripción	El Usuario Normal puede modificar su PIN (clave secreta de acceso a la tarjeta) por otro diferente, siempre y cuando la nueva clave respete las restricciones de seguridad y tamaño impuestas por el hardware (véase la sección de descripción de las tarjetas en el punto 3).
Precondiciones	Estar autenticado en el sistema como Usuario Normal.
Postcondiciones	El PIN del Usuario Normal es modificado por el nuevo suministrado.
Escenario Básico	• El usuario normal introduce el PIN antiguo. • El usuario normal introduce el PIN nuevo. • El PIN es modificado dentro del a tarjeta.
Escenario Alternativo	• Se introduce un PIN que no cumple las restricciones hardware (por ejemplo, no llega a la longitud mínima)

Tabla 25: Caso de Uso "Cambio de PIN de Usuario Normal"

Nombre	Cambio de Etiqueta - General
Actores	Usuario Normal y Tarjeta+Cryptoki.
Descripción	El Usuario Normal puede cambiar la etiqueta de cualquier objeto presente dentro de la tarjeta (incluidas las claves privadas). El tamaño máximo que se admitirá en la aplicación para las etiquetas será de 64 caracteres.
Precondiciones	Estar autenticado en el sistema como Usuario Normal. Existir al menos un objeto en la tarjeta.
Postcondiciones	La etiqueta del objeto en cuestión es modificada.
Escenario Básico	• Se selecciona el objeto al que se le quiere modificar la etiqueta. • Se suministra la nueva etiqueta para el objeto. • Se modifica la etiqueta del objeto.
Escenario Alternativo	No aplica.

Tabla 26: Caso de Uso "Cambio de Etiqueta - General"

Nombre	Cambio de ID - General
Actores	Usuario Normal y Tarjeta+Cryptoki.

Descripción	El Usuario Normal puede cambiar el identificador interno de cualquier objeto presente dentro de la tarjeta (incluidas las claves privadas). El tamaño máximo que se admitirá en la aplicación para los nuevos identificadores será de 64 caracteres.
Precondiciones	Estar autenticado en el sistema como Usuario Normal. Existir al menos un objeto en la tarjeta.
Postcondiciones	El ID del objeto en cuestión es modificado.
Escenario Básico	• Se selecciona el objeto al que se le quiere modificar su ID. • Se suministra el nuevo ID para el objeto (se convertirá a hexadecimal el texto ASCII suministrado). • Se modifica el ID del objeto.
Escenario Alternativo	No aplica.

Tabla 27: Caso de Uso "Cambio de ID - General"

Nombre	Firmar Fichero
Actores	Usuario Normal y Tarjeta+Cryptoki.
Descripción	Mediante esta funcionalidad el usuario podrá firmar cualquier fichero dentro del ordenador local en donde se ejecuta la aplicación. Para ello, utilizará alguna de las claves privadas que tiene en su tarjeta. El resultado será un fichero con la firma digital del fichero antes mencionado. La firma irá en formato ASCII.
Precondiciones	Estar autenticado en el sistema como Usuario Normal. Existir al menos una clave privada en la tarjeta.
Postcondiciones	Se obtiene un fichero con la firma digital.
Escenario Básico	• Se expresa la ruta el nombre del fichero que se desea firmar. • Se selecciona la clave privada con la que se va a firmar el fichero. • Se selecciona el mecanismo de firma. • Se suministra la ruta y el nombre del fichero donde se guardará la firma.
Escenario Alternativo	• Se selecciona el fichero para comprobar la firma. ○ El fichero o la ruta no existen.

Tabla 28: Caso de Uso "Firmar Fichero"

Nombre	Borrar Objetos - General
Actores	Usuario Normal y Tarjeta+Cryptoki.
Descripción	El usuario puede eliminar cualquier objeto dentro de la tarjeta (existen ciertas excepciones hardware para algunos modelos, como son los PINes en las tarjetas Starcos). En las tarjetas Ceres, se produce la peculiaridad de que, al borrar una clave privada, se elimina también su correspondiente clave pública.
Precondiciones	Estar autenticado en el sistema como Usuario Normal. Existir al menos un objeto en la tarjeta.
Postcondiciones	El objeto seleccionado desaparece de la tarjeta.
Escenario Básico	• Selecciona el objeto que desea eliminar. • El objeto es eliminado de la tarjeta.

Escenario Alternativo	• Que el objeto seleccionado sea algún tipo de objeto que no siga en sí mismo el estándar PKC#11 como sucede con los PINes en Starcos. En este caso el objeto no se borraría realmente, sólo quedaría fuera de uso para la sesión actual.

Tabla 29: Caso de Uso "Borrar Objetos - General"

Nombre	Generar Par de Claves
Actores	Usuario Normal y Tarjeta+Cryptoki.
Descripción	Esta funcionalidad permite generar claves públicas y privadas en el interior de la tarjeta. Se permitirá generar claves cuyo módulo sea de 1024 o 2048 bits, salvo en el caso de Starcos que sólo se permite la generación de claves de 1024 bits.
Precondiciones	Estar autenticado en el sistema como Usuario Normal.
Postcondiciones	Se genera una clave pública en el sistema. Se genera una clave privada en el sistema.
Escenario Básico	• Se selecciona el tamaño de la clave (1024 ó 2048) salvo en Satarcos. • Se suministra la etiqueta que tendrán las claves generadas. • Se generan las claves dentro de la tarjeta en forma de dos nuevos objetos, una clave pública y otra privada.
Escenario Alternativo	No aplica.

Tabla 30: Caso de Uso "Generar Par de Claves"

Nombre	Importar PKCS#12
Actores	Usuario Normal y Tarjeta+Cryptoki.
Descripción	Esta funcionalidad le permite al Usuario Normal importar una identidad digital dentro de la tarjeta. Este PKCS#12 contiene una clave pública, su clave privada y un certificado que sirve de envoltorio a la clave pública. Generalmente los PKCS#12 están protegidos por un cifrado simétrico y se requiere de una contraseña para poder importarlos. En caso de que no esté cifrado el PKCS#12, se suministrará una contraseña vacía.
Precondiciones	Estar autenticado en el sistema como Usuario Normal. Disponer de un fichero PFX con la identidad digital.
Postcondiciones	Se creará un certificado digital correspondiente con el contenido en el PFX. Se creará su clave pública. Se creará su clave privada.
Escenario Básico	• Se selecciona la ruta y el nombre de PFX. • Se suministra el password para el PFX (puede ser vacío). • Se suministra la etiqueta que tendrán los objetos generados en la tarjeta. • Se generan las claves y el certificado en la tarjeta correspondientes a los datos contenidos en el PFX.
Escenario Alternativo	• La ruta o el nombre del PFX es errónea. • La clave que encripta el PFX es incorrecta.

	• El PFX contiene claves con más bits de los soportados por las tarjetas (por ejemplo claves de 4096 bits).

Tabla 31: Caso de Uso "Importar PKCS#12"

Nombre	Importar Certificado
Actores	Usuario Normal y Tarjeta+Cryptoki.
Descripción	El usuario importa un certificado X.509 desde un fichero a la tarjeta, pudiendo asociar dicho certificado a alguna clave privada contenida en la tarjeta (enfocado a importar certificados de logon).
Precondiciones	Estar autenticado en el sistema como Usuario Normal. Tener un certificado X.509 en el ordenador.
Postcondiciones	Se almacena el certificado X.509 dentro de la tarjeta.
Escenario Básico	• Suministra la ruta y el nombre del certificado • Opcionalmente se selecciona la clave privada a la que se desea asociar el certificado • Se selecciona la etiqueta para el certificado. • Se crea el objeto certificado en la tarjeta.
Escenario Alternativo	• La ruta o el nombre del certificado son incorrectas.

Tabla 32: Caso de Uso "Importar Certificado"

Nombre	Logout Usuario Normal
Actores	Usuario Normal y Tarjeta+Cryptoki.
Descripción	El usuario cierra su sesión dentro de la tarjeta adquiriendo el rol de Usuario sin sesión.
Precondiciones	Estar autenticado en el sistema como Usuario Normal.
Postcondiciones	El usuario pierde el rol de Usuario Normal y adquiere el de Usuario sin sesión.
Escenario Básico	• El usuario realiza el logout. • El usuario adquiere el rol de Usuario sin sesión.
Escenario Alternativo	No aplica.

Tabla 33: Caso de Uso "Logout Usuario Normal"

Nombre	Cambio de ID - Públicos
Actores	Oficial de Seguridad y Tarjeta+Cryptoki.
Descripción	El Oficial de Seguridad puede cambiar el identificador interno de cualquier objeto público presente dentro de la tarjeta (excluidas las claves privadas). El tamaño máximo que se admitirá en la aplicación para los nuevos identificadores será de 64 caracteres.
Precondiciones	Estar autenticado en el sistema como Oficial de Seguridad. Existir al menos un objeto en la tarjeta.
Postcondiciones	El ID del objeto en cuestión es modificado.
Escenario Básico	• Se selecciona el objeto al que se le quiere modificar su ID. • Se suministra el nuevo ID para el objeto (se convertirá a

	hexadecimal el texto ASCII suministrado). • Se modifica el ID del objeto.
Escenario Alternativo	No aplica.

Tabla 34: Caso de Uso "Cambio de ID - Públicos"

Nombre	Cambio de Etiqueta - Públicas
Actores	Oficial de Seguridad y Tarjeta+Cryptoki.
Descripción	El Oficial de Seguridad puede cambiar la etiqueta de cualquier objeto público dentro de la tarjeta (excluidas las claves privadas). El tamaño máximo que se admitirá en la aplicación para las etiquetas será de 64 caracteres.
Precondiciones	Estar autenticado en el sistema como Oficial de Seguridad. Existir al menos un objeto en la tarjeta.
Postcondiciones	La etiqueta del objeto en cuestión es modificada.
Escenario Básico	• Se selecciona el objeto al que se le quiere modificar la etiqueta. • Se suministra la nueva etiqueta para el objeto. • Se modifica la etiqueta del objeto.
Escenario Alternativo	No aplica.

Tabla 35: Caso de Uso “Cambio de Etiqueta - Públicas”

Nombre	Borrar Objetos - Públicos
Actores	Oficial de Seguridad y Tarjeta+Cryptoki.
Descripción	El Oficial de Seguridad puede eliminar cualquier objeto público de la tarjeta (existen ciertas excepciones hardware para algunos modelos, como son los PINes en las tarjetas Starcos). En las tarjetas Ceres, se produce la peculiaridad de que, al borrar una clave privada, se elimina también su correspondiente clave pública.
Precondiciones	Estar autenticado en el sistema como Oficial de Seguridad. Existir al menos un objeto en la tarjeta.
Postcondiciones	El objeto seleccionado desaparece de la tarjeta.
Escenario Básico	• Selecciona el objeto que desea eliminar. • El objeto es eliminado de la tarjeta.
Escenario Alternativo	• Que el objeto seleccionado sea algún tipo de objeto que no siga en sí mismo el estándar PKC#11, como sucede con los PINes en Starcos. En este caso, el objeto no se borraría realmente, sólo quedaría fuera de uso para la sesión actual.

Tabla 36: Caso de Uso "Borrar Objetos - Públicos"

Nombre	Cambiar PIN Oficial de Seguridad
Actores	Oficial de Seguridad y Tarjeta+Cryptoki.
Descripción	El Oficial de Seguridad puede modificar su PIN (clave secreta de acceso a la tarjeta) por otra diferente, siempre y cuando la nueva clave respete las restricciones de seguridad y tamaño impuestas por el hardware (véase la

	sección de descripción de las tarjetas en el punto 3).
Precondiciones	Estar autenticado en el sistema como Oficial de Seguridad.
Postcondiciones	El PIN del Oficial de Seguridad es modificado por el nuevo suministrado.
Escenario Básico	• El usuario normal introduce el PIN antiguo. • El usuario normal introduce el PIN nuevo. • El PIN es modificado dentro del a tarjeta.
Escenario Alternativo	• Se introduce un PIN que no cumple las restricciones hardware (por ejemplo, no llega a la longitud mínima)

Tabla 37: Caso de Uso "Cambiar PIN Oficial de Seguridad"

Nombre	Logout Oficial de Seguridad
Actores	Oficial de Seguridad y Tarjeta+Cryptoki.
Descripción	El Oficial de Seguridad cierra su sesión dentro de la tarjeta adquiriendo el rol de Usuario sin sesión.
Precondiciones	Estar autenticado en el sistema como Oficial de Seguridad.
Postcondiciones	El usuario pierde el rol de Oficial de Seguridad y adquiere el de Usuario sin sesión.
Escenario Básico	• El usuario realiza el logout. • El usuario adquiere el rol de Usuario sin sesión.
Escenario Alternativo	No aplica.

Tabla 38: Caso de Uso "Logout de Oficial de Seguridad"

Nombre	Desbloquear PIN de Usuario Normal
Actores	Oficial de Seguridad y Tarjeta+Cryptoki.
Descripción	El Oficial de Seguridad puede desbloquear el PIN del Usuario Normal una vez que se bloqueó por introducir varias veces consecutivas un PIN erróneo (generalmente son 3 los intentos, pero esto depende del hardware de las tarjetas, véase punto 3).
Precondiciones	Estar autenticado en el sistema como Oficial de Seguridad.
Postcondiciones	El PIN del Usuario Normal vuelve a ser operativo.
Escenario Básico	• Suministra un nuevo PIN para ser usado por el Usuario Normal (acción de desbloqueo). • El nuevo PIN “desbloqueado” queda activado para ser usado por el Usuario Normal.
Escenario Alternativo	• El nuevo PIN suministrado no cumple las restricciones hardware de la tarjeta (como, por ejemplo, la longitud mínima).

Tabla 39: Caso de Uso "Desbloquear PIN de Usuario Normal"

5 – DISEÑO

Una vez se ha realizado el análisis y se conoce perfectamente qué funcionalidad debe tener la aplicación a desarrollar y, además, se tiene información sobre las restricciones y limitaciones de las tarjetas, se puede acometer el diseño de la aplicación para que dé cobertura a todo lo tratado en el apartado anterior.

Debido a que se acordó que la aplicación sería desarrollada en C, el diseñó debe estar enfocado a una aplicación basada en módulos (componentes) los cuales ofrecen unas funcionalidades usadas por el resto.

Agrupando las diferentes funcionalidades en contenedores más grandes (módulos), se consigue crear entidades que tienen un denominador común: por ejemplo, trabajar sobre un mismo tipo de objeto, hacer frente a funcionalidades comunes o trabajar con recursos compartidos.

Para la realización de la aplicación, se decidió agrupar las funcionalidades que actúan sobre un recurso o datos comunes en módulos, de modo que cada uno de los módulos intenta dar solución (funcional) a una serie de requisitos funcionales del sistema. Por tanto, habrá: un módulo encargado de la interacción con el Cryptoki de la tarjeta, otro encargado de la gestión de Certificados X.509 y objetos similares del disco. Habrá módulos que trabajen ciertos tipos de TAD (como listas, por ejemplo) y módulos encargados de dar soporte a otros módulos ofreciendo funcionalidades genéricas que no tienen finalidad por sí mismas.

Para plasmar de la mejor forma posible las ideas que se vayan planteando en el diseño, se hará uso de diferentes tipos de diagramas UML, que permitan ver de forma gráficas y las hagan más entendibles. Se usarán diagramas de componentes para definir los diferentes módulos presentes en la aplicación. Estos diagramas están enfocados al diseño orientado a objetos; pero, si entendemos un módulo como un ente que nos proporciona una funcionalidad, vemos que se puede representar perfectamente una aplicación estructurada con un diagrama de componentes. También se usarán diagramas de actividad para representar los algoritmos diseñados para dar solución a los diferentes requisitos planteados en el análisis.

5.1 – MÉTODO DE DISEÑO

Como ya se ha mencionado, se ha seguido un diseño basado en componentes, donde cada componente se centra en satisfacer una serie de funcionalidades con un matiz común (generalmente basadas en el tipo de elementos gestionados).

En primer lugar, se identifican las funcionalidades a realizar. Esto ya se hizo en el análisis (véase, por ejemplo, el apartado de Casos de uso punto 4.5). Una vez se tienen las funcionalidades, se agrupan por el carácter distintivo.

Una vez se han obtenidos los módulos que realizan las funcionalidades mencionadas en el análisis, hay que plantearse si, para la realización de dichas funciones, se requiere de alguna capa que dé soporte a dicha funcionalidad, pero que, debido a sus características, no sea una capa funcional, sino enfocada a que la funcionalidad sea realizada de una forma más simple y cómoda. De esta parte, aparecen nuevos módulos enfocados a ofrecer funcionalidades comunes para terceros módulos, tales como: listas enlazadas u otras estructuras de datos, funcionalidades auxiliares, etc.

Una vez se han definido todos los módulos y las relaciones entre ellos, es momento de definir el comportamiento de las funcionalidades que contienen dichos módulos. En este apartado, se define el funcionamiento de cada una de las funcionalidades descritas en el análisis. Es una descripción a bajo nivel, ya no se centra el foco sobre lo que se hace, sino en el cómo se hace.

Finalmente, hay que tener en cuenta las diferentes características y capacidades de las tarjetas para hacer diseños ad hoc si fuera necesario para algún modelo concreto de tarjeta usado. Esto se traduce en el uso del preprocesador (ya que como se mencionó en el análisis la implementación se realizará en ANSI C). De este modo, puede haber diferentes matices de funcionamiento para ciertas funcionalidades en función del modelo de tarjeta al que esté destinada la compilación de la aplicación.

En el siguiente punto, se van a definir los componentes diseñados en la aplicación, las funcionalidades que contienen, tipos abstractos de datos, etc.

5.2 – DESCRIPCIÓN DE LOS COMPONENTES

Los componentes diseñados se componen, en general, de un fichero .c y otro .h. El .h es el fichero que define los tipos de datos abstractos y enumera las funcionalidades disponibles para el componente (la interfaz pública al exterior). Por otro lado, el .c es el encargado de implementar la funcionalidad expresada en el .h. En otras palabras, el .h sería una representación del análisis (las cosas que se hacen) y el .c sería la implementación de las mismas (el cómo se hacen).

Hay algunas excepciones: por ejemplo, el módulo principal del programa (donde está el main de la aplicación) no tiene .h, ya que se limitará a invocar las funcionalidades ofrecidas por otros. También está el caso del Cryptoki, que se compone de cuatro ficheros .h donde se representan las funciones y tipos de datos y la DLL concreta que implementa dichas funcionalidades.

Siguiendo los planteamientos anteriormente mencionados, se han diseñado los siguientes módulos para la aplicación:

- El módulo principal, que es el encargado de gestionar las llamadas a las funcionalidades. Es el que implementa el método main y genera la interfaz textual para el usuario. Sólo tiene un fichero .c que será programa.c.
- El módulo PKCS#11, que se encarga de realizar las secuencias de llamadas correctas al Cryptoki para obtener la funcionalidad esperada. Se compone de smartcard.c y smartcard.h.
- El módulo OpenSSL, es el encargado de gestionar la lectura de ficheros externos que contiene elementos PKI, como son los certificados X.509 y los perfiles PKCS#12. Su nombre se debe a que usa la API de OpenSSL para poder interactuar con dichos elementos. Se compone de los ficheros openssl.c y openssl.h.
- El módulo Lista, se encarga de implementar el TAD lista enlazada que da soporte a los objetos PKCS#11 leídos de la tarjeta. De éste modo, se tiene acceso a manejadores e información básica desde la lista recuperada de la tarjeta y ahorrando constantes llamadas a la tarjeta en sí. Se compone de los ficheros listaObjetos.c y listaObjetos.h.
- El módulo Mecanismos, se encarga de gestionar la parte del Cryptoki relacionada con los mecanismos presentes en la tarjeta y para qué se utilizan. No hace llamas al Cryptoki, sino que sirve de soporte a PKCS#11 para facilitarle estas tareas. Se compone de los ficheros mecanismos.c y mecanismos.h.
- El módulo Soporte. Éste módulo ofrece funciones auxiliares enfocadas a la traducción de tipos de datos (como fechas, por ejemplo), a formatear datos para ser mostrados por pantalla y traducción de errores y operaciones similares.

Esta capa es usada por muchos de los anteriores módulos. Se compone de los ficheros soporte.c y soprote.h.

- El módulo Cryptoki. Es la interfaz del sistema para invocar la funcionalidad ofrecida por la tarjeta. Este módulo no es implementado en este proyecto, simplemente, es usado para realizar las operaciones solicitadas. Se compone por un lado de cuatro ficheros de cabecera: cryptoki.h, pkcs11.h, pkcs11t.h y pkcs11f.h. Por otro lado, está la DLL que es donde se implementa la funcionalidad descrita. Esta DLL varía de una tarjeta a otra, pero el resto del módulo (y de todos los módulos anteriores) permanece invariante.

El diagrama de despliegue de la aplicación sería el siguiente:

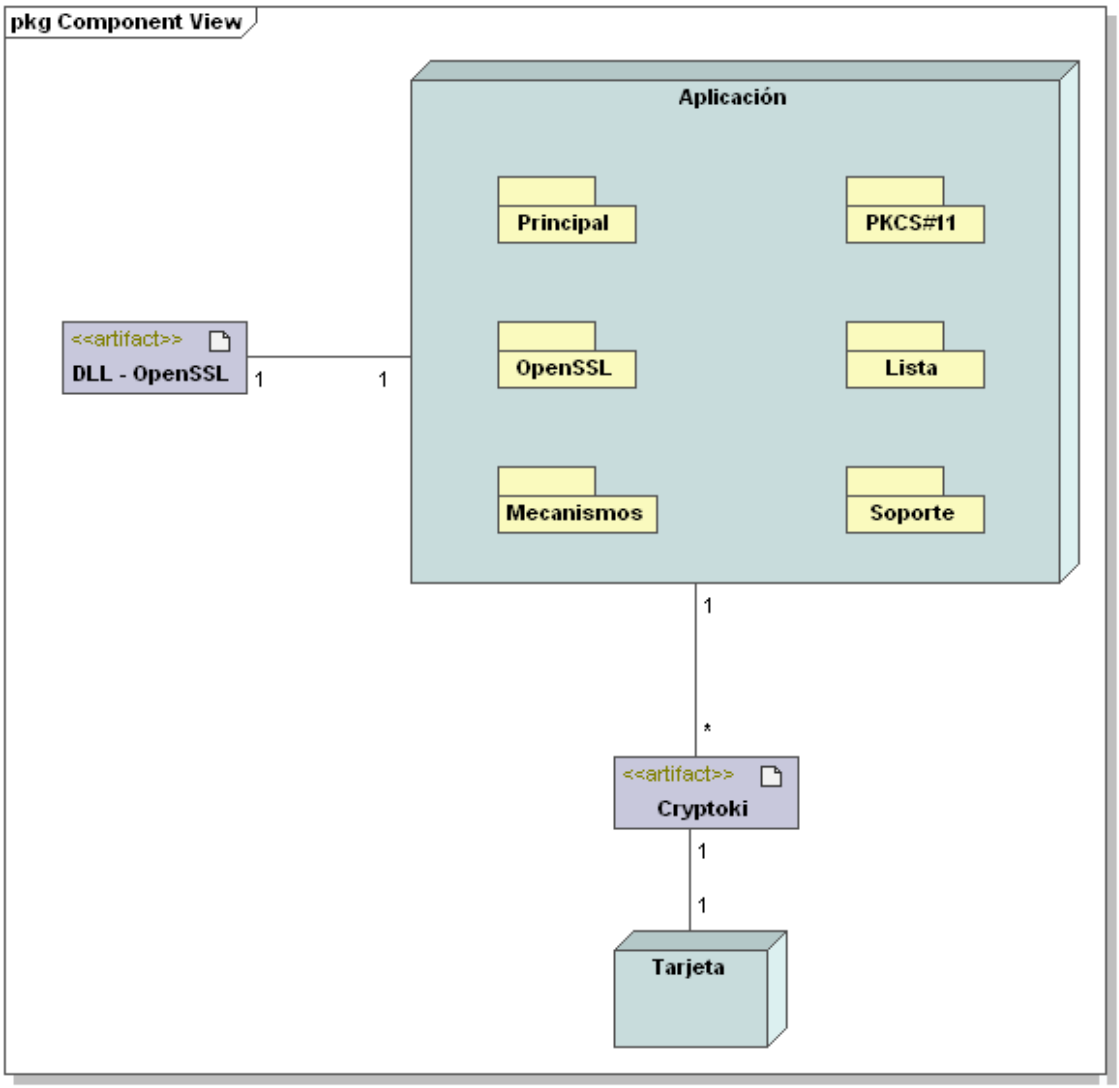

Ilustración 24: Diagrama de despliegue

En el diagrama, se ve un nodo Aplicación, que representa la aplicación que contiene los seis elementos que la constituyen (los implementados en este proyecto). No se han incluido las relaciones entre ellos, ya que esto se hará en el diagrama de componentes. Se observa también el artefacto Cryptoki, que hace referencia a la DLL de la tarjeta y a sus ficheros de cabecera. La relación de la aplicación con los Cryptokis es de 1 a N, ya que la aplicación hará uso de varias DLLs de tarjetas (aunque sólo hará uso de una en cada compilación).

Finalmente, cada Crytptoki se relaciona con una tarjeta, la suya, a la que invocará la funcionalidad que se le solicite.

Ahora, se va a ver el diagrama de componentes del sistema, donde sí se verán las relaciones entre los mismos.

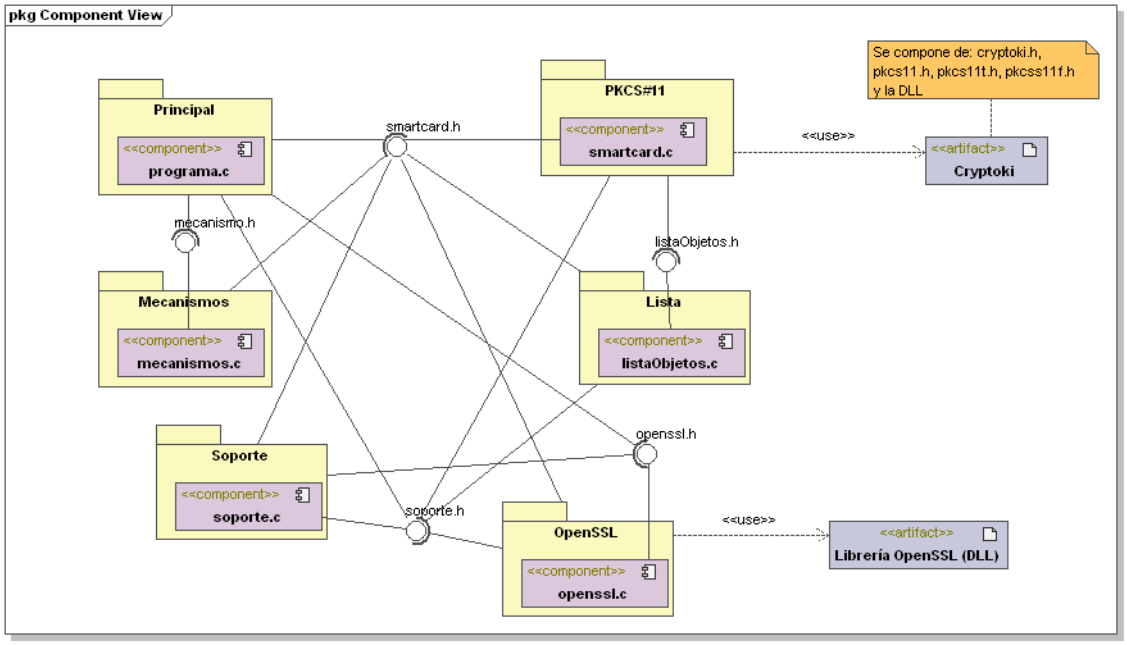

Ilustración 25: Diagrama de componentes

Para hacer más legible el diagrama, se va a expresar un listado con descripciones de las relaciones que se establecen entre los distintos componentes.

Componente	Implementa	Usa
Principal		• smartcard.h • mecanismos.h • openssl.h • soporte.h

Componente	Implementa	Usa
PKCS#11	• smartcard.h	• soporte.h • Cryptoki
Mecanismos	• mecanismos.h	• smartcard.h
Lista	• listaObjetos.h	• smartcard.h
Soporte	• soporte.h	• smartcard.h • openssl.h
OpenSSL	• openssl.h	• soprote.h • smartcard.h • DLL de OpenSSL

Tabla 40: Relaciones entre Componentes

A continuación, se hará una descripción, un poco más en profundidad, de cada uno de los módulos planteados en el diseño. Después de ver para qué se usa cada módulo y ver sus características, se hará un diseño de todas las funcionalidades expresadas en el análisis.

5.2.1 – Módulo Principal

Este es el módulo de ejecución del programa y, por tanto, incluye el main de la aplicación. Su funcionalidad es la de proveer de una capa de llamadas a funciones, de modo que, cuando se invoca una, se generará una secuencia más compleja de llamadas a funciones de otros módulos para así poder realizar la funcionalidad solicitada. El módulo principal está encarnado en el fichero programa.c.

Principal también es el encargado de poner en funcionamiento la interfaz en consola y la gestión de toda la entrada/salida de las acciones y sus resultados.

De smartcard.h obtiene toda la funcionalidad relacionada con acciones a realizar por la tarjeta. De mecanismos.h obtiene el soporte necesario para obtener y gestionar los mecanismos hardware presentes en la tarjeta. De openssl.h invoca la funcionalidad relativa a la gestión de objetos PKI existentes en disco (como certificados, PKCS#12, etc.). Finalmente, de soport.h importa la funcionalidad relacionada con la salida a pantalla (limpiar, mostrar ciertos tipos de datos, etc.).

Principal define una serie de constantes:

- `#define NoUser 0`: Representa al Usuario sin sesión.
- `#define NormalUser 1`: Representa al Usuario Normal.
- `#define SecureOfficer 2`: Representa al Oficial de Seguridad.

Además, principal usa o implementa las siguientes variables globales:

- `extern CK_SLOT_ID lector;`: Representa el ID del lector que se usa durante le ejecución de la aplicación (sólo se permite el uso de un lector durante una ejecución).
- `extern CK_SESSION_HANDLE sesion;`: Representa el manejador de la sesión. Esta información le sirve al Cryptoki para asociar peticiones a la tarjeta con una aplicación concreta. Esta es la variable más importante de toda la aplicación.
- `extern objeto *listaDeObjetos;`: Es el punto de acceso a la lista de objetos con información de todos los objetos a los que se tiene acceso dentro de la tarjeta.
- `extern CK_MECHANISM_TYPE_PTR listaMecanismos;`: Esta es la lista (array dinámico) que permite saber qué mecanismos tiene disponibles la tarjeta y, también, a través de esta estructura, poder saber para qué se utiliza cada uno de esos mecanismos.
- `CK_ULONG tamMecanismo;`: Representa el tamaño del array dinámico anterior (para saber cuántos mecanismos hay).
- `CK_ULONG tipoUsuario = NoUser;`: Representa al tipo de usuario que está trabajando en la sesión actual, para facilitar en todo momento ciertas acciones sin tener que invocar a la tarjeta. Por defecto, se inicializa como `NoUser`, ya que sin haber hecho login ese es el rol que se tiene al iniciar la aplicación.

5.2.1.1 – Funciones del Módulo Principal

No se va a hacer un definición rigurosa de cómo están diseñadas las funciones del módulo principal ya que son funciones que no realizan acciones en sí mismas, sólo sirven para invocar llamadas a terceros, como al módulo PKCS#11 que sí implementa funcionalidad propiamente dicha.

Las funciones del Módulo Principal se encargan, a la hora de realizar las invocaciones de las funcionalidades, de garantizar que el entorno que rodea a esas invocaciones es seguro, dicho de otro modo, realiza todas las comprobaciones oportunas para garantizar que todos los datos, objetos, etc., están presentes y son correctos para la realización de la funcionalidad. Para realizar esas comprobaciones, se realizan invocaciones a las siguientes funciones:

- `int elementosTipo (CK_OBJECT_CLASS tipo, objeto *lista)`. Definida en el Módulo de Soporte. Su finalidad es decir cuántos elementos hay en la lista (tiene información de todos los objetos accesibles de

la tarjeta) del tipo especificado. Se utiliza, por ejemplo, para comprobar si existe alguna clave privada a la hora de firmar un fichero.

- `CK_BBOOL paramValor (char *entrada, int *num, int min, int max)`. Esta función sirve para validar parámetros leídos por teclado y obtener su valor entero. La función retornará verdadero si la cadena de entrada contenía un número válido. El valor convertido se retorna en num. min y max representan el rango válido de valores, de modo que si el número introducido es válido, pero está fuera de ese margen, se retorna falso. Esta función se utiliza, por ejemplo, para comprobar que los parámetros introducidos son correctos a la hora de introducir un número que representa a un objeto.
- `CK_OBJECT_HANDLE manejadorPosicion (objeto *lista, int posicion, CK_OBJECT_CLASS tipo)`. Esta función le sirve al módulo principal para obtener los manejadores de los objetos de la lista de objetos. Estos objetos (o mejor dicho, sus manejadores) son los que serán usados para "utilizar" ciertas funcionalidades que los requiere, como, por ejemplo, una clave pública a la hora de comprobar una firma digital.

Por tanto, el módulo principal genera la interfaz textual, implementa todas las funciones envoltorio cuya finalidad es asegurar que los datos a utilizar son válidos y, finalmente, invocar funciones de otros módulos que sí realizan la funcionalidad solicitada. Con esto se consigue un desacoplamiento que otorga un grado mayor de mantenibilidad a la aplicación, ya que los cambios en cómo se realiza la funcionalidad son independientes de los datos que esa funcionalidad maneja. Cada una de las funciones del módulo principal se corresponde con una opción mostrada en la interfaz textual de modo que, cuando se elige una operación concreta, lo que se hace es invocar a su función de Principal para que la gestione.

Para representar las comprobaciones que se realizan en las funciones del módulo principal, se va a plantear el diagrama de actividad de una función significativa del módulo Principal. Una significativa es la de firma, ya que en ella se seleccionan ficheros, objetos, mecanismos, etc. También se mostrará las acciones relacionadas con el mantenimiento de la interfaz textual que realizan las funciones del Módulo Principal.

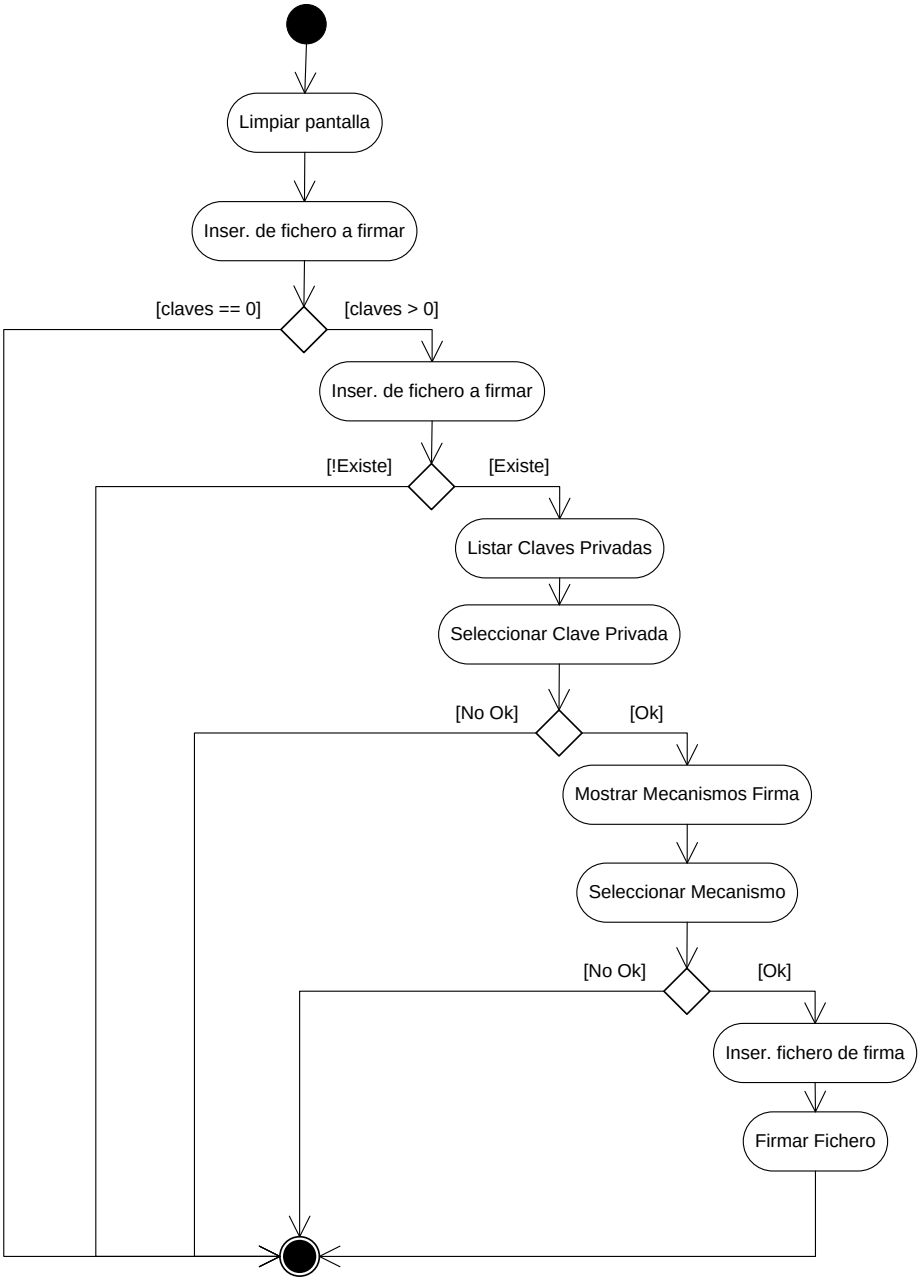

Ilustración 26: Ejemplo función Módulo Principal "Firma de Fichero"

5.2.1.2 – Interfaz textual de usuario

El funcionamiento básico de la interfaz es mediante la inserción de números que se corresponden con operaciones a realizar o con objetos a seleccionar. De este modo, la interfaz muestra una serie de opciones (que podrían ser las funciones a realizar, modalidades a elegir u objetos que seleccionar) identificadas por un número, de modo que el usuario debe escribir el número asociado a lo que desea hacer o seleccionar y pulsar Enter para realizar la acción.

En función de los datos suministrados, la interfaz mostrará resultados o errores de modo que el usuario pueda tener una idea correcta de lo que ha sucedido al realizar las operaciones solicitadas.

Lo primero que se nos muestra, al arrancar el programa, es el menú de selección del lector a utilizar para usar la tarjeta. Se mostrarán todos los lectores de tarjetas reconocidos en el ordenador que tengan introducida una SmartCard. En caso de no haber ninguno, se mostrará un error. A cada lector se le asigna un número y se espera a que el usuario elija ese número para utilizar ese lector concreto. Un ejemplo sería:

```
LISTA DE LECTORES CON TOKEN PRESENTE

0 - Aladdin Knowledge Systems Ltd.

Elija el lector:
```

Una vez se ha elegido el lector concreto, se accede a la pantalla principal del menú de funcionalidad. En ella se muestran todas las acciones que puede realizar un usuario. El menú principal es el siguiente:

```
MENU PRINCIPAL

 0 - Salir
 1 - login
 2 - logout
 3 - Desbloquear PIN
 4 - Cambiar PIN
 5 - Listar elementos
 6 - Mostrar certificado
 7 - Mostrar clave publica
 8 - Cambio ID
 9 - Cambiar etiqueta
10 - Borrar elementos
11 - Generar par de claves RSA
12 - Lista de mecanismos
13 - Ver info del lector
14 - Ver info de la tarjeta
15 - Numero aleatorio
16 - Firmar fichero
17 - Comprobar firma de fichero
18 - Crear peticion de certificado
19 - Importar certificado
20 - Asociar certificado a clave privada
21 - Importar PKCS#12
22 - Exportar certificado
```

Cada opción muestra, a su vez, submenús donde se permite seleccionar diferentes opciones u objetos necesarios para acometer la funcionalidad seleccionada. Son, en general, menús bastante similares así que sólo se mostrará a modo de ejemplo el de firma digital.

```
Nombre del fichero a firmar: C:\bar.emf
SLECCION DE LA CLAVE
LISTADO DE ELEMENTOS DE LA SMARTCARD
 0 - CLAVE PRIVADA:
      Etiqueta: RSA
      ID: 31:32:36:35:39:37:36:33:31:34
Eleccion de clave: 0
SELECCION DEL MECANISMO DE FIRMADO

0 - DES_MAC
1 - DES_MAC_GENERAL
2 - DES3_MAC
3 - DES3_MAC_GENERAL
4 - AES_MAC
5 - AES_MAC_GENERAL
6 - RSA_PKCS
7 - RSA_X_509
8 - MD5_RSA_PKCS
9 - SHA1_RSA_PKCS
10 - MD5_HMAC_GENERAL
11 - MD5_HMAC
12 - SHA_1_HMAC_GENERAL
13 - SHA_1_HMAC

Eleccion del mecanismo: 9
Nombre del fichero para almacenar la firma: C:\firma.txt
FIRMA DIGITAL DE FICHERO
El fichero se firmo satisfactoriamente...
Pulse enter para continuar...
```

Como se ve en la captura de consola, en primer lugar, se nos pide un nombre y ruta de fichero a firmar (que es introducido por teclado). Si el fichero existe, se muestra la lista de claves privadas (ya que son las necesarias para firmar), con su etiqueta y su ID. Todas los objetos tienen un número a la izquierda (en este caso sólo uno con un 0), ese número es el identificador a utilizar para seleccionar el objeto en cuestión, tal y como se hace en la Elección de clave.

Una vez se ha seleccionado el objeto, y éste es válido, se selecciona el mecanismo de firma, para lo que nuevamente se nos muestra un listado identificado por números con todos los mecanismos de firma soportados por la tarjeta (en este

caso una Aladdin eToken). Una vez seleccionado un mecanismo válido, se nos pide el nombre del fichero y su ruta donde queremos que se almacene la firma digital.

Al finalizar la operación, se nos informa de qué ha sucedido (con un error por ejemplo). En este ejemplo, la firma se realizó bien y así se nos comunica.

Finalmente, se va a mostrar el resultado de visualizar los elementos de una SmartCard, concretamente de un DNIe en el cual ya se ha hecho login (por lo que aparecerán objetos privados). Para todos los elementos aparece el ID interno de la lista de objetos (número de la izquierda), la etiqueta y el ID (identificador que relaciona elementos). En el caso de los certificados, también aparecerá el número de serie, salvo las etiquetas todos los datos han sido omitidos por tratarse de un DNIe real):

```
LISTADO DE ELEMENTOS DE LA SMARTCARD
 0 - CLAVE PUBLICA:
     Etiqueta: KpuFirmaDigital
     ID: 46:38:36…
 1 - CLAVE PUBLICA:
     Etiqueta: KpuAutenticacion
     ID: 41:38:36…
 2 - CLAVE PRIVADA:
     Etiqueta: KprivFirmaDigital
     ID: 46:38:36…
 3 - CLAVE PRIVADA:
     Etiqueta: KprivAutenticacion
     ID: 41:38:36…
 4 - CERTIFICADO:
     Etiqueta: CertFirmaDigital
     ID: 46:38:36…
     Numero Serie: …
 5 - CERTIFICADO:
     Etiqueta: CertAutenticacion
     ID: 41:38:36…
     Numero Serie: …
 6 - CERTIFICADO:
     Etiqueta: CertCAIntermediaDGP
     ID: 53:38:36…
     Numero Serie: …
```

5.2.2 – Módulo PKCS#11

Este módulo se compone de los ficheros smartcard.c y su interfaz smartcard.h. Su finalidad es realizar invocaciones al Cryptoki para así poder interactuar con la tarjeta (como se observó en el diagrama de despliegue).

Este módulo está enfocado a realizar las sucesiones de llamadas necesarias al Cryptoki para realizar las operaciones concretas que se mostraron en la interfaz (sólo aquellas que tienen que ver con PKCS#11). smartcar.c no realiza nunca invocaciones sueltas a funciones del Cryptoki, sino que genera funciones de alto nivel enfocadas a realizar una tarea generalmente muy grande que requiere de un gran número de invocaciones. De este modo se simplifica todo mucho y hace que sea increíblemente fácil usar las SmartCards (a través de este módulo no del Cryptoki directamente). A modo de ejemplo, con este módulo, se puede firmar un fichero con tan sólo una invocación.

Este módulo utiliza al Módulo de Soporte para obtener ciertas funcionalidades, como son: la de traducción de errores devueltos por el Cryptoki o ciertas macros auxiliares de uso general (como contar el número de posiciones de un array). En la descripción de las funcionalidades, se verá más detalladamente todos los usos que hace PKCS#11 de dicho módulo.

Las constantes y variables globales definidas por smartcard.h dentro del Módulo PKCS#11 (o usadas desde el exterior) son:

- ```
  #if defined (CERES) || defined (ALADDIN)
       #define MAXLENPASS 16
       #define MINLENPASS 8
  #endif
  #if defined (STARCOS)
       #define MAXLENPASS 8
       #define MINLENPASS 4
  #endif
  ```
  En este bloque de código se observa la primera discriminación por tipo de tarjeta. Si la compilación se realiza para tarjetas Ceres o Aladdin, la longitud máxima del PIN será de 16 bytes (caracteres) y la mínima 8. En caso de tratarse de Starcos, la máxima será 8 y la mínima 4.
- `#define BLOQUE 64`: Este es el tamaño del buffer usado para lectura y escritura de ficheros y, por extensión, en general, para cualquier buffer usado en la aplicación.
- `#define TAMFICH 256`: Este valor se usa para definir el tamaño máximo reservado para nombres de fichero (incluida su ruta de acceso).
- `#define MAXELEMENTOS 1024`: Esta constante define el número máximo de elementos de una SmartCard capaz de gestionar la aplicación. Lo normal es que haya menos de 10 elementos.
- `#define MAXLONGID 10`: Esta constante define la longitud de los identificadores de objetos que sean generados por la aplicación (se usará el formato de tiempo en milisegundos con 10 caracteres).
  ```

- `#define MAXLENPUK 32`: Esta constante define el número bytes (caracteres) máximos para el PIN del Oficial de Seguridad.
- `CK_SESSION_HANDLE sesion;`: Esta es la variable más importante de toda la aplicación. Es un acceso global al identificador de la sesión (el manejador de sesión). Este identificador es requerido para casi todas las operaciones que se realicen sobre el Cryptoki.
- `CK_SLOT_ID lector;`: Esta variable es un punto de acceso global al identificador (manejador) del lector. Todas las funcionalidades que requieran referenciar al lector usarán este valor.
- `objeto *listaDeObjetos;`: Es el punto de acceso global a la lista de objetos de la tarjeta. Podría haberse definido en el Módulo Lista, pero hacía más sencillo el manejo de llamadas el incluirlo aquí. De esta lista se obtendrá acceso a las propiedades de todos los objetos visibles en la tarjeta en un momento dado.
- `CK_MECHANISM_TYPE_PTR listaMecanismos;`: Es un punto de acceso al array de mecanismos de la tarjeta. Esta variable es mantenida, principalmente, por el módulo auxiliar de Mecanismos. Su finalidad es almacenar información sobre los mecanismos soportados en la tarjeta (por medio de sus identificadores).

A continuación, se van a definir todas las funcionalidades implementadas en este módulo. Para ello, se presentará un diagrama UML de actividad, donde se verá el funcionamiento del algoritmo diseñado y se realizarán los comentarios que se consideren oportunos para complementar la información suministrada por el diagrama. También se incluirán fragmentos de código siempre y cuando sean importantes o necesarios para facilitar la comprensión de la funcionalidad. Todas las funciones implementadas en el Módulo de PKCS#11 retornan `CKR_OK` si se ejecutaron bien o por el contrario el primer error obtenido de invocar al Cryptoki si hubo algún problema.

5.2.2.1 – Iniciar Sistema

```
CK_RV iniciarSistema ();
```

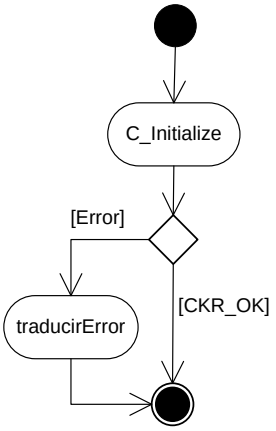

Ilustración 27: iniciarSistema

Esta función permite inicializar el Cryptoki. No tiene un efecto práctico, pero es imprescindible inicializar el Cryptoki antes de usarlo. Invoca la función `C_Initialize` del Cryptoki para inicializarlo. En caso de no inicializarse, no se podría utilizar ninguna llamada en general. Esta función sólo debe ser llamada una única vez, el resto de ocasiones produciría error.

5.2.2.2 – Finalizar Sistema

```
CK_RV finalizarSistema ();
```

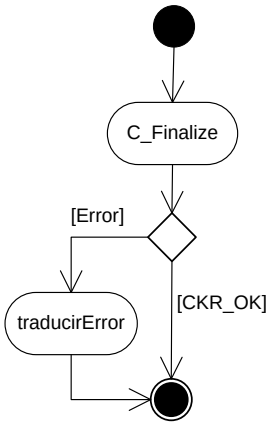

Ilustración 28: finalizarSistema

Esta función es la opuesta a `iniciarSistema ()`. Su finalidad es terminar de usar el Cryptoki. Cuando ya no se va a utilizar más la tarjeta y se va a cerrar la aplicación, es conveniente invocar esta función para liberar adecuadamente el Cryptoki. De este modo el Cryptoki podría liberar todos los recursos asociados a la ejecución. La liberación del Cryptoki se realiza invocando a `C_Finalize ()`.

5.2.2.3 – Obtener número de Slot del Sistema

```
CK_RV obtenerNumeroSlot (CK_ULONG_PTR numero);
```

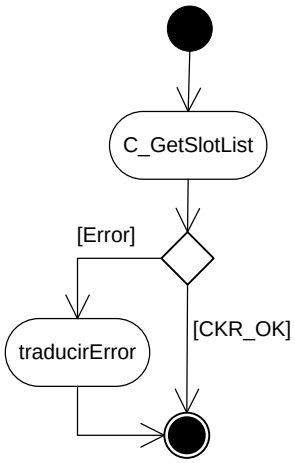

Ilustración 29: obtenerNumeroSlot

Esta función invoca `C_GetSlotList (CK_TRUE, NULL_PTR, numero);` para obtener en el puntero `numero` la cantidad de lectores disponibles en el sistema. El primer argumento `CK_TRUE` quiere decir que sólo liste aquellos lectores que tienen una SmartCard insertada. El segundo argumento es `NULL_PTR` y representa que no queremos almacenar en ningún sitio el listado de identificadores, sólo queremos obtener el número. Esto es necesario hacerlo así, ya que sin el número de lectores no sabemos cuanta memoria habría que reservar para almacenar esos identificadores.

5.2.2.4 – Obtener Listado de Slots del Sistema

```
CK_RV obtenerListadoSlots (CK_SLOT_ID_PTR ptrSlot, CK_ULONG numero);
```

Esta función comparte el mismo diagrama y funcionamiento que la anterior (`obtenerNumeroSlot`). La diferencia es que esta función se invocará pasando en el puntero `numero` la cantidad de lectores disponibles en el sistema. La invocación a Cryptoki se hace `C_GetSlotList (CK_TRUE, ptrSlot, &numero);` de modo que el segundo argumento ya no es `NULL_PTR` sino `ptrSlot`, que es la ruta (con memoria prereservada) donde queremos almacenar el listado de identificadores de los lectores con SmartCard presente.

5.2.2.5 – Información de Slot

```
CK_RV infoSlot (CK_SLOT_ID IDLector);
```

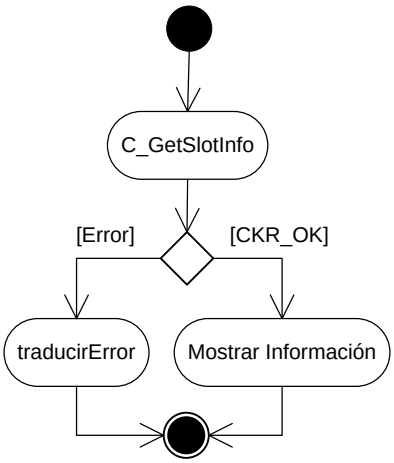

Ilustración 30: infoSlot

Esta función permite obtener información de un lector concreto a partir de su manejador `IDLector`. La invocación al Cryptoki se realiza con `C_GetSlotInfo (IDLector, &infoSlot);` donde el primer argumento es el manejador del lector del que queremos la información y el segundo es una estructura `CK_SLOT_INFO infoSlot` que es donde se va almacenar la información obtenida.

5.2.2.6 – Información Extendida de Slot

Esta función es idéntica a la anterior, sólo se diferencia en que a la hora de obtener la información en la estructura `CK_SLOT_INFO` se recuperan más datos a fin de dar al usuario una información más completa sobre el lector. La información obtenida es el nombre del lector, la versión del firmware, la versión de hardware y el ID del fabricante. Muchos de estos datos son genéricos y no correctos de cada lector.

5.2.2.7 – Información del Token

```
CK_RV infoToken ();
```

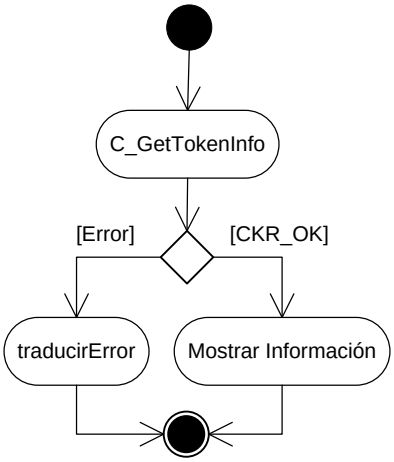

Ilustración 31: infoToken

Esta función obtiene información asociada al token (SmartCard) insertada dentro del lector. Hay que tener en cuenta que, una vez se ha elegido un lector en el sistema, éste pasa a ser una variable global (descrita en este mismo Módulo) por lo que esta función no requiere parámetros. La invocación al Crytptoki se realiza con `C_GetTokenInfo (lector, &tokInf);` siendo lector el manejador del lector y `tokInf` una estructura de tipo `CK_TOKEN_INFO`, que es donde se almacenará la información asociada a la tarjeta, que es: Nombre de la tarjeta, versión del firmware, versión hardware y el ID del fabricante de la tarjeta (los mismos que para el lector).

5.2.2.8 – Abrir Sesión

```
CK_RV abrirSesion (CK_FLAGS flags);
```

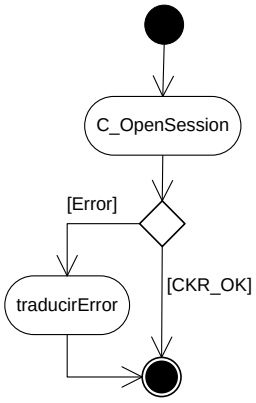

Ilustración 32: abrirSesion

La invocación a abrirSesion, permite abrir una sesión dentro de la tarjeta. Es el primer paso para poder obtener los objetos de la misma, de hecho con la sesión abierta ya se tiene acceso a todos los objetos públicos de la SmartCard. Es necesario tener una sesión abierta (y su manejador) para poder operar sobre los objetos y

también es necesaria para poder hacer login de usuario. Al Cryptoki se le invoca con `C_OpenSession (lector, flags, (CK_VOID_PTR) &application, NULL_PTR, &sesion);` donde `lector` es el manejador del lector, `flags` tiene el valor `CKF_SERIAL_SESSION | CKF_RW_SESSION` que representan sesión en serie (el primer argumento de OR. Es una imposición del Cryptoki, sin él daría error) y que es una sesión de lectura escritura. Todas las sesiones que se abren en la aplicación son siempre de lectura escritura. Si todo va bien en `sesión`, se obtendrá el manejador de la sesión que se almacenará como global.

5.2.2.9 – Cerrar Sesión

```
CK_RV cerrarSesion ();
```

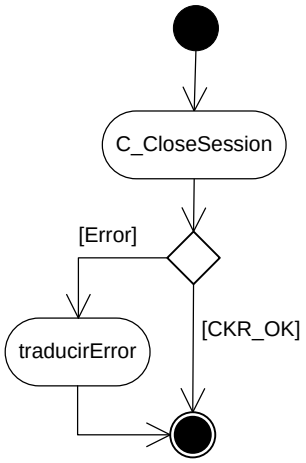

Ilustración 33: cerrarSession

Esta es la función encargada de cerrar una sesión previamente abierta. No tiene ninguna complejidad, simplemente se invoca al Cryptoki `con C_CloseSession (sesion);` donde `sesion` es el manejador de la sesión obtenido en `abrirSesion`.

5.2.2.10 – Login

```
CK_RV cerrarSesion (CK_USER_TYPE usuario, CK_UTF8CHAR_PTR userPIN);
```

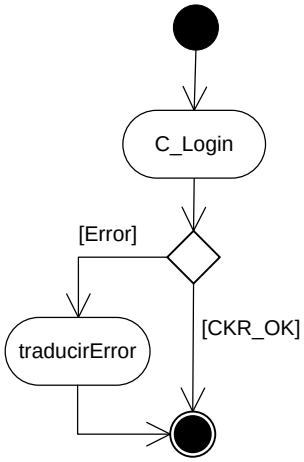

Ilustración 34: login

`Login` sirve para cambiar el rol del usuario de Usuario sin sesión (como se definió en el análisis en el punto 4) al de Usuario Normal o a Oficial de Seguridad. Al cambiar de rol, se adquieren nuevos privilegios que permiten realizar nuevas operaciones (todas ya descritas). Para hacer login, se invoca al Cryptoki con `C_Login (sesion, usuario, userPIN, strlen (userPIN));` donde `sesion` es el manejador de la sesión, usuario es el tipo de usuario y puede ser `CKU_USER` para el Usuario Normal o `CKU_SO` para el Oficial de Seguridad. `userPIN` contiene el PIN del usuario y el último argumento es la longitud del PIN, ya que, como se admite cualquier carácter para el PIN, el carácter `NULL` no marca el final del PIN y se requiere, por tanto, pasar su longitud. Si se hace `login,` cuando ya se había hecho previamente, se produce un error. Hay que hacer un `logout` antes de volver a hacer un nuevo `login` de usuario.

5.2.2.11 – Logout

```
CK_RV logout ();
```

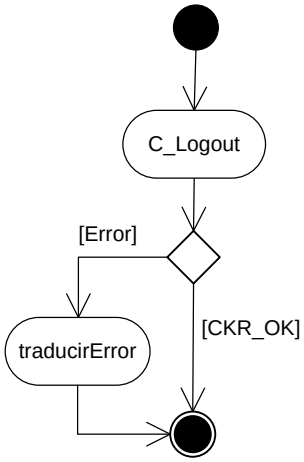

Ilustración 35: logout

Esta función sirve para perder el rol de uno de los dos usuarios con privilegio y adquirir nuevamente el de Usuario sin sesión. Si se hace un `logout` sin un `login` correcto previo el resultado será un error. Para hacer `logout` se invoca al Cryptoki con `C_Logout (sesion);` donde `sesion` es el manejador de la sesión actual.

5.2.2.12 – Cambio de PIN

```
CK_RV cambioPIN (CK_UTF8CHAR_PTR oldPIN, CK_UTF8CHAR_PTR newPIN);
```

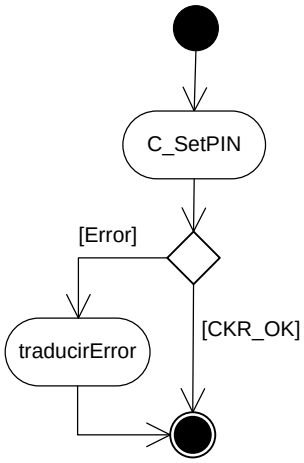

Ilustración 36: CambioPIN

Esta función permite cambiar el PIN del usuario que está autenticado en el sistema. Por tanto, el PIN modificado será el del rol que hizo el `login`, porque es necesario haber hecho `login` para poder cambiar el PIN. Para modificar el PIN se requiere del PIN antiguo correcto y del nuevo. El PIN nuevo deberá cumplir los requisitos hardware de la tarjeta (como el de la longitud), en caso contrario, producirá

un error el cambio de PIN. Se invoca la Cryptoki con `C_SetPIN (sesion, oldPIN, strlen (oldPIN), newPIN, strlen (newPIN));` donde se le pasa la sesión actual, los PINes antiguo y nuevo y sus respectivas longitudes.

5.2.2.13 – Recorrer Elementos

CK_RV recorrerElementos ();

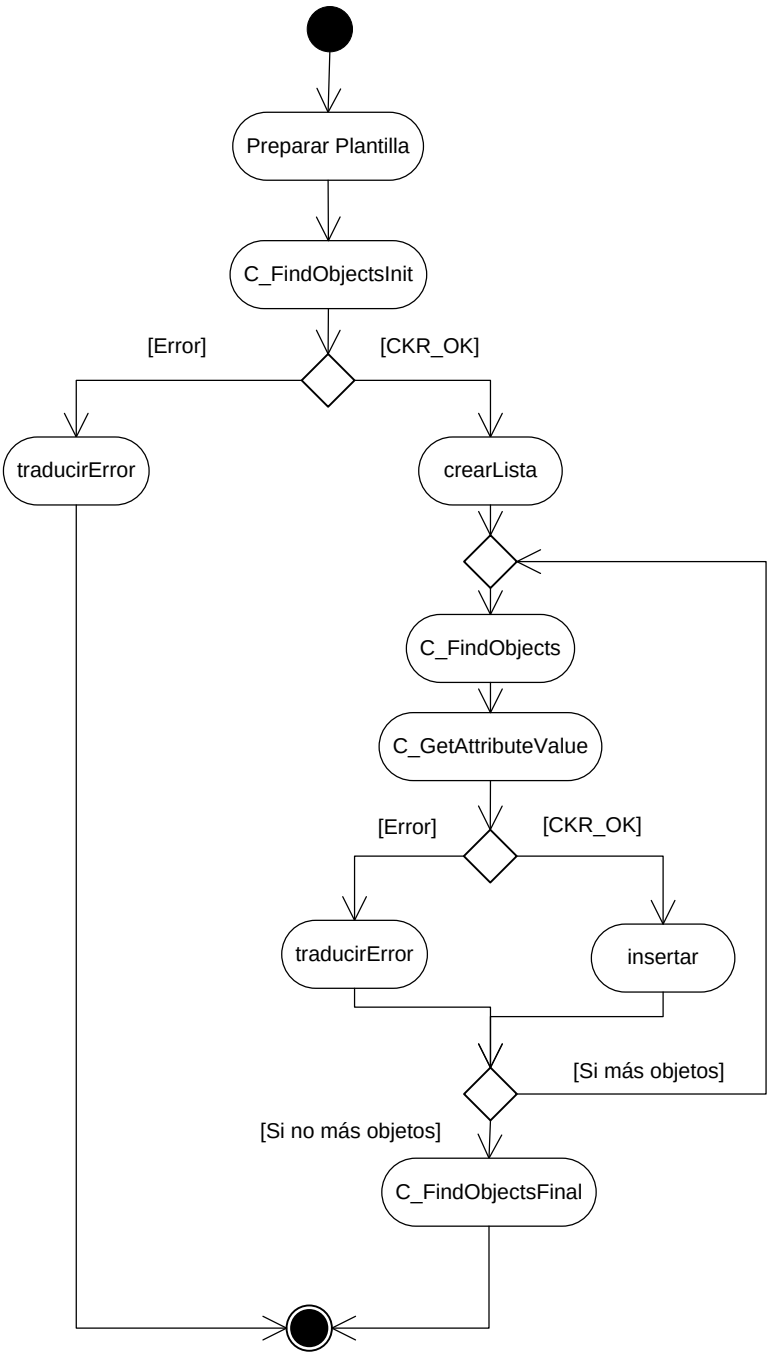

Ilustración 37: recorrerElementos

Para recorrer los elementos de la tarjeta, el primer paso que hay que realizar es preparar la plantilla con los datos que queremos obtener de los objetos a los que tenemos acceso de la tarjeta. La plantilla no es más que un array de tipo `CK_ATTRIBUTE` (tipo, valor y longitud del valor). La plantilla que se va a usar es:

```
CK_ATTRIBUTE plantillaTipo [] = {
     {CKA_CLASS, &clase, sizeof (clase)}
};
```

Sólo contiene el atributo clase, ya que es el único valor que queremos obtener de los objetos para insertarlos en la lista (el manejador se obtiene más adelante). A continuación, se parapara la búsqueda con la llamada `C_FindObjectsInit (sesion, NULL_PTR, 0);` donde `sesion` es el manejador de la sesión, `NULL_PTR` indica que no queremos filtrar ningún objetos (que se recuperen todos los que son accesibles). A continuación, se crea (o regenera) la lista de objetos.

Si todo va bien, se invoca la obtención de datos del objeto con `C_FindObjects (sesion, &objeto, 1, &contObjeto);` donde `sesion` es el manejador del a sesión, objeto es el puntero donde se recogerá el manejador del objeto actual encontrado, el 1 representa que los objetos se obtendrán de uno en uno y `contObjecto` es el puntero que apunta a la cantidad de objetos recuperados en esta pasada del bucle (como se dice que como máximo 1, recuperará 1 ó 0).

Una vez se tiene el manejador del objeto recuperado, se puede invocar la obtención de sus atributos, concretamente lo expresados en la plantilla antes descrita. Para ello, se invoca al Cryptoki `con C_GetAttributeValue (sesion, objeto, plantillaTipo, NUM_ELEMTOS (plantillaTipo));` donde se le pasa el manejador de la sesión, `plantillaTipo`, que es la plantilla donde se quieren recuperar los valores y un último parámetro con el número de elementos del array (en ese caso será uno).

Si los atributos son obtenidos de forma correcta, se inserta un nodo en la lista de objetos. Se repite el bucle hasta que en la llamada `C_FindObjects` el puntero `contObjecto` valga 0, que significa que ya no se han podido recuperar más objetos.

Cuando termina la ejecución de la función se tiene una lista con información de los tipos de objetos contendidos en la tarjeta y sus manejadores.

5.2.2.14 – Generar Números Aleatorios

```
CK_RV generarNumeroAleatorio (CK_ULONG tam);
```

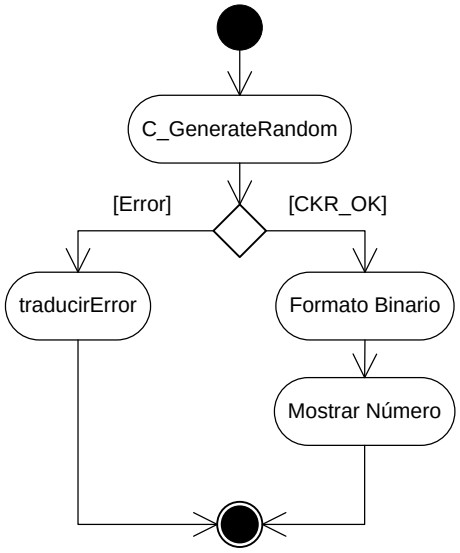

Ilustración 38: generarNumeroAleatorio

Para generar números aleatorios con la tarjeta es necesario invocar al Cryptoki con `C_GenerateRandom (sesion, random, tam);` donde `sesion` es el manejador de la sesión actual, `random` es un array donde se almacenará el número obtenido y `tam` es la longitud (en bytes) del número a generar. La memoria de `random` tiene que estar previamente reservada para el correcto funcionamiento.

Una vez el número es generado, se formatea a binario utilizando los operadores de desplazamiento de bits (<<). Una vez que se ha convertido el número a bits (1s y 0s), es mostrado en ese formato binario.

5.2.2.15 – Crear Certificado

```
CK_RV crearCertificado (char * etiqueta, char cert [], int lCert, char * sn, int lSN, char *issuer, int lI, char *subject, int lS, CK_DATE nB, CK_DATE nA, time_t tiempo);
```

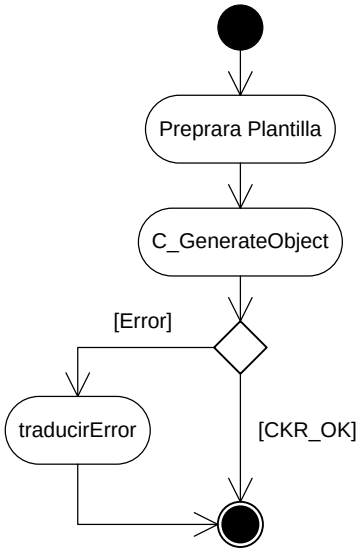

Ilustración 39: crearCertificado

Crear un certificado dentro de la tarjeta, desde el punto de vista de llamadas al Cryptoki, es muy sencillo, ya que sólo requiere una llamada. La complejidad radica en qué datos se le pasan a la llamada (y como están formateados dichos datos). La plantilla (y datos asociados) usados para generar los certificados es (se incluye el preprocesamiento necesario para adaptar la plantilla de la mejor forma a cada tarjeta en función de sus diferencias):

```
CK_OBJECT_CLASS clase = CKO_CERTIFICATE;
#ifndef ALADDIN
     CK_CERTIFICATE_TYPE tCert = CKC_X_509;
     CK_BBOOL false = CK_FALSE;
#endif
CK_BBOOL true = CK_TRUE;
CK_BYTE ID [MAXLONGID];
memset (ID, 0, MAXLONGID);
sprintf (ID, "%d", (int)tiempo);
CK_ATTRIBUTE certificateTemplate [] = {
     {CKA_CLASS, &clase, sizeof (clase)},
     {CKA_LABEL, etiqueta, strlen(etiqueta)},
     {CKA_VALUE, cert, lCert},
     #ifndef ALADDIN
          {CKA_SERIAL_NUMBER, sn, lSN},
          {CKA_CERTIFICATE_TYPE, &tCert, sizeof (tCert)},
          {CKA_ISSUER, issuer, lI},
          {CKA_PRIVATE, &false, sizeof (false)},
     #endif
     {CKA_SUBJECT, subject, lS},
     {CKA_ID, ID, sizeof (ID)},
     #ifdef STARCOS
```

```
            {CKA_START_DATE, &nB, sizeof (nB)},
            {CKA_END_DATE, &nA, sizeof (nA)},
        #endif
        {CKA_TOKEN, &true, sizeof (true)}
};
```

En primer lugar, se expresa que el objeto que vamos a crear es de tipo certificado (`CKO_CERTIFICATE`), luego, se define que el tipo concreto de certificado es `CKC_X_509`, salvo en el caso de Aladdin que no admite esa descripción de subtipos de certificados.

En la plantilla, se expresa la clase, la etiqueta (con sus respectivas longitudes) y, luego, en el campo `CKA_VALUE`, se inserta la codificación DER completa del Certificado X.509. El siguiente bloque no es válido para tarjetas Aladdin. En ese apartado, se expresa el número de serie del certificado, el tipo, el emisor y, finalmente, se expresa que el certificado es público.

Al final, se expresa el sujeto propietario del certificado y el ID del objeto. El ID siempre es la hora local tomada en milisegundos y convertida en cadena ASCII de 10 caracteres. La conversión es literal, de modo que el dígito 0 se convierte en el ASCII (48) (representación en formato carácter del "0") y así sucesivamente. Esta regla de asignación de ID que garantiza que nunca se repitan a no ser que eso sea lo que se busca. Esta regla de asignación se utiliza en todas las creaciones de objetos (claves públicas y privadas, certificados, etc.).

Una vez que se ha rellenado la plantilla, se invoca la funcionalidad del Cryptoki por medio de `C_CreateObject (sesion, certificateTemplate, NUM_ELEMTOS (certificateTemplate), &manejadorObjeto);` donde sesion es el manejador de la sesión actual, certificateTempalte es la plantilla anteriormente rellenada. Luego se expresa el número de atributos que contiene esa plantilla y, finalmente, en el puntero, `manejadorObjeto` se nos retorna el manejador que se le asignó al objeto recién creado.

5.2.2.16 – Crear Clave Privada

```
CK_RV crearClavePrivada (char * etiqueta, char *d, int
ld, char *dmp1, int ldmp1, char *dmq1, int ldmq1, char *e,
int le, char *iqmp, int liqmp, char *n, int ln, char *p,
int lp, char *q, int lq, time_t tiempo);
```

El diagrama de actividad de crear una clave privada es exactamente igual al de crear certificados. La diferencia, en sí, con esa función, radica en la plantilla que se le

suministra al sistema (y los datos que contiene dicha plantilla). A continuación, se expondrá la plantilla utilizada para crear estas claves. Hay que tener en cuenta que esta función está pensada para suministrar los datos de la clave desde el exterior (por ejemplo, un PKCS#12) y no para que la tarjeta cree la clave en sí.

```
CK_OBJECT_CLASS clase = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_RSA;
 CK_RV rv;
CK_BBOOL true = CK_TRUE;
CK_BYTE ID [MAXLONGID];
memset (ID, 0, MAXLONGID);
sprintf (ID, "%d", (int)tiempo);
CK_ATTRIBUTE keyeTemplate [] = {
     {CKA_CLASS, &clase, sizeof (clase)},
     {CKA_KEY_TYPE, &keyType, sizeof (keyType)},
     {CKA_LABEL, etiqueta, strlen(etiqueta)},
     CKA_ID, ID, sizeof (ID)},
     {CKA_PRIVATE, &true, sizeof (true)},
     {CKA_SENSITIVE, &true, sizeof (true)},
     {CKA_DECRYPT, &true, sizeof (true)},
     {CKA_SIGN, &true, sizeof (true)},
     {CKA_TOKEN, &true, sizeof (true)},
     {CKA_MODULUS, n, ln},
     CKA_PUBLIC_EXPONENT, e, le},
     {CKA_PRIVATE_EXPONENT, d, ld},
     {CKA_PRIME_1, p, lp},
     {CKA_PRIME_2, q, lq},
     {CKA_EXPONENT_1, dmp1, ldmp1},
     {CKA_EXPONENT_2, dmq1, ldmq1},
     {CKA_COEFFICIENT, iqmp, liqmp}
};
```

Como se ve en la plantilla (teniendo en cuenta los parámetros que se le han pasado a la invocación de esta función) se suministran todos los elementos que componen una clave privada (los primos, los módulos, los exponentes, etc.), así como sus respectivas longitudes. Cabe destacar que se marca como privada la futura clave (`CKA_PRIVATE, &true, sizeof (true)`) así como que es sensible (por medio de `CKA_SENSITIVE`).

Una vez se ha definido la plantilla, se invoca al Cryptoki con `C_CreateObject (sesion, keyeTemplate, NUM_ELEMTOS (keyeTemplate), &manejadorObjeto);` donde sesion es el manejador de la sesión, keyeTempalte es la plantilla anteriormente descrita. Luego se expresa su

longitud (en número de elementos del array) y, finalmente, la invocación nos retornará el manejador del objeto recién creado en la dirección de memoria de manejadorObjeto.

5.2.2.17 – Crear Clave Pública

```
     CK_RV crearClavePublica (char * etiqueta, char *e, int
le, char *n, int ln, time_t tiempo);
```

El procedimiento de crear una clave pública es idéntico al de la clave privada (para los ocasos en los que ya se dispone de los valores de la clave importados del exterior). En primer lugar, se crea la plantilla con la siguiente estructura:

```
CK_OBJECT_CLASS clase = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_RSA;
CK_RV rv;
CK_BBOOL true = CK_TRUE;
CK_BYTE ID [MAXLONGID];
CK_ATTRIBUTE keyeTemplate [] = {
     {CKA_CLASS, &clase, sizeof (clase)},
     {CKA_KEY_TYPE, &keyType, sizeof (keyType)},
     {CKA_LABEL, etiqueta, strlen(etiqueta)},
     {CKA_ID, ID, sizeof (ID)},
     {CKA_ENCRYPT, &true, sizeof (true)},
     {CKA_VERIFY, &true, sizeof (true)},
     {CKA_TOKEN, &true, sizeof (true)},
     {CKA_MODULUS, n, ln},
     {CKA_PUBLIC_EXPONENT, e, le}
};
```

La plantilla es similar, salvando que los campos de los valores de la clave son distintos (sólo se trabaja con la clave pública y el exponente). También difieren en el tipo de elemento (aquí es `CKO_PUBLIC_KEY`, que también es otro cambio lógico y evidente). En esta ocasión, la clave no es privada ni tampoco sensible (es más, ni tan siquiera están expresados en la plantilla).

Una vez está rellenada la plantilla con los datos adecuados, se invoca al Cryptoki con `C_CreateObject (sesion, keyeTemplate, NUM_ELEMTOS (keyeTemplate), &manejadorObjeto);` (es la misma llamada que en el caso de la clave privada), donde los parámetros significan lo mismo que en la invocación de la función de generar clave privada.

5.2.2.18 – Cambiar ID de objetos

```
CK_RV cambioID (CK_OBJECT_HANDLE objetoBase,
CK_BYTE_PTR nuevoID, int lNID);
```

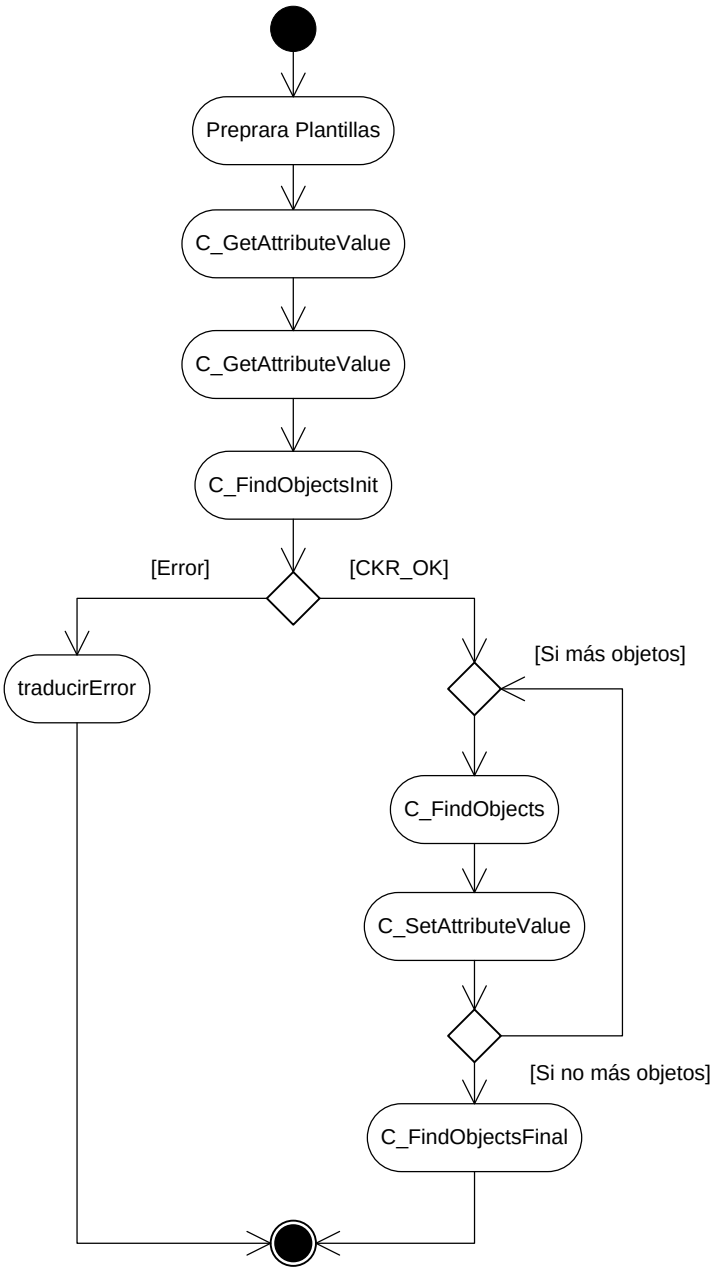

Ilustración 40: cambioID

El cambio de ID tiene una peculiaridad, no sólo cambia el ID del objeto seleccionado, en su lugar cambia el ID de todos los objetos que comparten ID con el objeto seleccionado. Esto se hace así, ya que si cambiamos el ID de un objeto y no se hace lo mismo con el resto de objetos que tienen su mismo ID (por ejemplo, un

certificado con sus claves, o claves entre sí), esos objetos quedarán huérfanos y serán inservibles en muchos aspectos (por ejemplo, los CSP de Windows se basan en buscar objetos que comparten ID).

La forma de trabajar de esta función se basa, en primer lugar, en obtener el ID del objeto seleccionado (por medio de su manejador). En primer lugar, se invoca `C_GetAttributeValue (sesion, objetoBase, modificable, NUM_ELEMTOS (modificable));` donde modificable es un plantilla con valor `NULL_PTR` y longitud 0. Esto supone que la función nos retornará, en longitud, el tamaño del ID. Una vez tenemos el tamaño, volvemos a invocar a la misma función; pero, tras haber cambiado la plantilla, asignando la longitud y la memoria a los atributos (cosa que antes se desconocía). En esa segunda invocación, es cuando obtenemos realmente el ID del objeto (dato solicitado en la plantilla).

A continuación, se inicia una búsqueda con `C_FindObjectsInit (sesion, modificable, NUM_ELEMTOS (modificable));`. Pasar la plantilla modificable con los datos obtenidos de `C_GetAttributeValue` supone que sólo buscará objetos que coincidan con dicha plantilla, en otras palabras, buscará sólo aquellos objetos que compartan el mismo ID (que es la información que contiene la plantilla modificable).

Finalmente, se entra en un while que se repetirá mientras sigan encontrándose objetos en la tarjeta que compartan el ID. Los objetos serán obtenidos (o mejor dicho sus manejadores) por medio de la invocación `C_FindObjects (sesion, &objetoMod, 1, &contObjeto);` (esta llamada ya fue explicada en la funcionalidad de este Módulo referida a recorrer objetos punto 5.2.1.13). Para cada uno se esos objetos se invoca `C_SetAttributeValue (sesion, objetoMod, modificada, NUM_ELEMTOS (modificada));` donde modificada es una plantilla que contiene el nuevo ID y su longitud. El resultado será que el objeto referenciado por `objetoMod` cambiará su ID al expresado por `modificada`.

Cuando el while finaliza, se invoca a `C_FindObjectsFinal (sesion);` para finalizar la búsqueda y poder concluir el proceso de cambios de ID en cascada.

5.2.2.19 – Cambiar etiqueta de objetos

```
CK_RV cambioEtiqueta (CK_BYTE_PTR label, CK_ULONG lLabel, CK_OBJECT_HANDLE manejador);
```

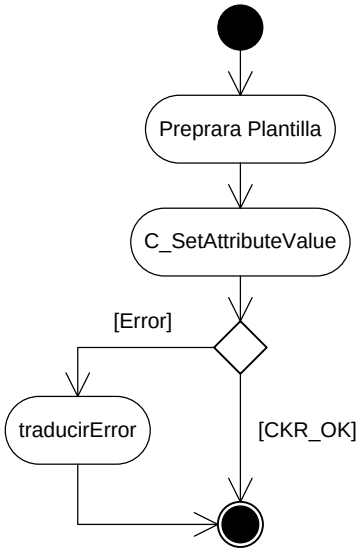

Ilustración 41: cambioEtiqueta

El funcionamiento de la modificación de etiquetas es muchos más simple que el de modificar identificadores, ya que las etiquetas son campos descriptivos sin utilidad dentro de la tarjeta y por tanto, no habrá que propagarlas.

En primer lugar, se genera una plantilla que contendrá el valor de la nueva etiqueta y su longitud. A continuación, se modificará el valor del objeto seleccionado (el valor de su etiqueta) por medio de la siguiente invocación del Cryptoki `C_SetAttributeValue (sesion, manejador, modificada, NUM_ELEMTOS (modificada));`.

5.2.2.20 – Obtener longitud de la clave pública

CK_RV obtenerLongitudClavePublica (CK_OBJECT_HANDLE manejador, CK_ULONG_PTR longitud);

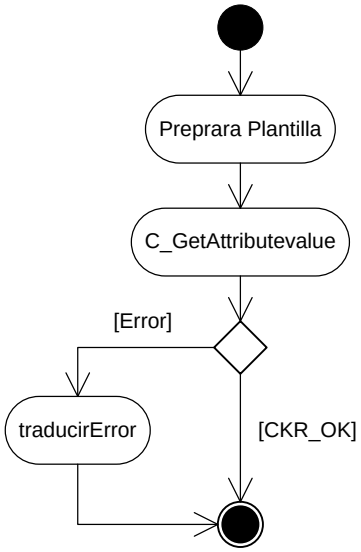

Ilustración 42: obtenerLongitudClavePublica

Esta función es muy simple, solamente genera una plantilla de consulta con el atributo `CKA_MODULOS_BITS`, de modo que al realziar la consulta al Cryptoki con `C_GetAttributeValue (sesion, manejador, plantilla, NUM_ELEMTOS (plantilla));` obtendremos en dicha plantilla el valor entero que representa los bits del módulo de la clave pública (que generalmente serán 1204 o 2048 bits).

La utilidad de esta función es poder conocer las longitudes que deben tener las firmas digitales que se van a verificar con la clave pública. Por ejemplo, una clave de tamaño 1024 bits espera una firma para la versificación de 128 bytes, en caso de ser mayor o menor, no se podría verificar.

5.2.2.21 – Obtener longitud de la clave privada

```
CK_RV obtenerLongitudClavePrivada (CK_OBJECT_HANDLE manejador, CK_ULONG_PTR longitud);
```

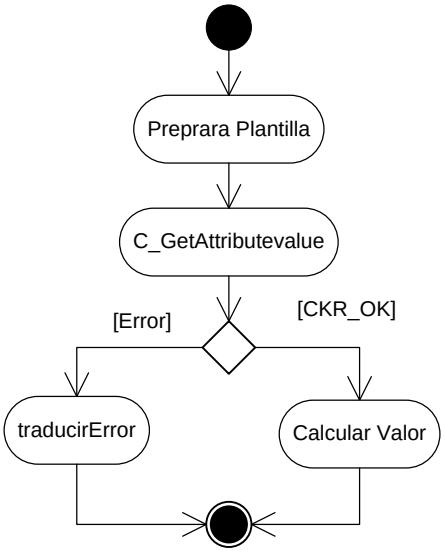

Ilustración 43: obtenerlongitudClavePrivada

El procedimiento de obtener la longitud de una clave privada es muy similar al de la clave pública. La diferencia radica en que las claves privadas no tienen el atributo `CKA_MODULOS_BITS` por lo que hay que consultar el atributo `CKA_MODULUS` con la llamada `C_GetAttributeValue (sesion, manejador, plantilla, NUM_ELEMTOS (plantilla));`. Esta llamada nos retornará la longitud de dicho atributo, ya que el módulo es un atributo de longitud variable (un puntero a una cadena) por lo que la primera llamada a `C_GetAttributeValue` retorna la longitud del atributo, para que, tras los arreglos oportunos, se pueda obtener su valor con una nueva llamada. Una vez tenemos al longitud (retornada en bytes), lo multiplicamos por 8 para obtener la longitud en bits y se retorna en la dirección de memoria correspondiente de la invocación el resultado de la operación.

5.2.2.22 – Eliminar Objetos

```
CK_RV destruirObjeto (CK_OBJECT_HANDLE objeto);
```

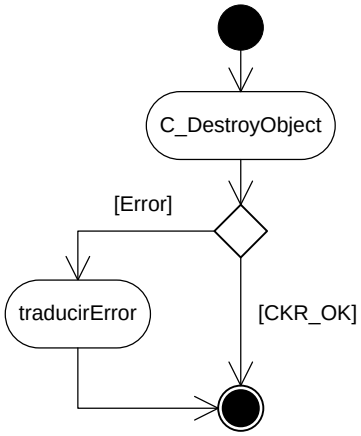

Ilustración 44: destruirObjeto

Destruir un objeto de la tarjeta es muy sencillo. Sólo tenemos que suministrar el manejador del objeto que queremos destruir e invocar del `Cryptoki C_DestroyObject (sesion, objeto);`. Esto supone que tenemos acceso al objeto (de no tener acceso no podríamos tener su manejador). Como ya se mencionó, esta función tiene diferencias de comportamiento entre los diferentes Cryptoki de las diferentes tarjetas, como es el caso de Ceres, en la cual, al borrar una clave privada se elimina también la clave pública asociada.

5.2.2.23 – Crear Par de Claves

```
CK_RV crearParDeClaves (CK_UTF8CHAR_PTR etiqueta, int longEtiqueta, CK_ULONG bitsModulo);
```

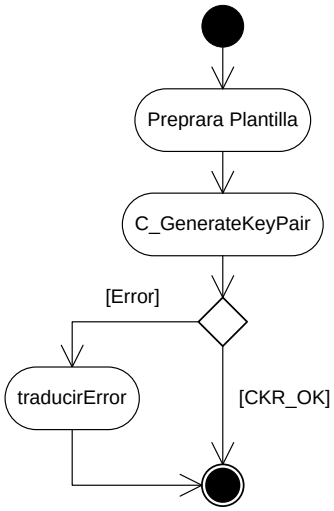

Ilustración 45: crearParDeClaves

Esta función, al igual que todas las de generar objetos dentro de la tarjeta, es muy sencilla de invocar, pero su complejidad radica en generar una plantilla consistente y en que contenga los datos adecuados para la correcta generación del par de claves. Las plantillas utilizadas (una para la clave pública y otra para la privada) son:

```
CK_MECHANISM mecanismo = {
     CKM_RSA_PKCS_KEY_PAIR_GEN, NULL_PTR, 0
};
CK_OBJECT_CLASS clasePub = CKO_PUBLIC_KEY;
CK_OBJECT_CLASS clasePri = CKO_PRIVATE_KEY;
CK_KEY_TYPE tipoClave = CKK_RSA;
CK_BYTE exponente [] = {0x01, 0x00, 0x01};
CK_BBOOL cierto = CK_TRUE;
CK_BYTE ID [MAXLONGID];
CK_ATTRIBUTE plantillaPublica [] = {
     {CKA_CLASS, &clasePub, sizeof (clasePub)},
     {CKA_LABEL, etiqueta, longEtiqueta},
     {CKA_MODULUS_BITS, &bitsModulo, sizeof (bitsModulo)},
     {CKA_KEY_TYPE, &tipoClave, sizeof (tipoClave)},
     {CKA_TOKEN, &cierto, sizeof (cierto)},
     {CKA_ID, ID, MAXLONGID},
     {CKA_PUBLIC_EXPONENT, exponente, sizeof (exponente)}
};
CK_ATTRIBUTE plantillaPrivada [] = {
     {CKA_CLASS, &clasePri, sizeof (clasePri)},
     {CKA_LABEL, etiqueta, longEtiqueta},
     {CKA_TOKEN, &cierto, sizeof (cierto)},
     {CKA_ID, ID, MAXLONGID},
     {CKA_PRIVATE, &cierto, sizeof (cierto)},
     {CKA_SENSITIVE, &cierto, sizeof (cierto)},
     {CKA_KEY_TYPE, &tipoClave, sizeof (tipoClave)}
};
```

En primer lugar, se expresa el mecanismo de la acción a realizar con CKM_RSA_PKCS_KEY_PAIR_GEN que representa la generación de pares de claves.

Para la plantilla de la clave pública, son destacables los atributos `CKA_MODULUS_BITS` que representa el tamaño del módulo (por extensión también representa en tamaño de la clave) y el valor de `CKA_PUBLIC_EXPONENT`, que es el exponente de la clave. Hay que mencionar que, en general, los módulos PKCS#11 de las tarjetas no generan un módulo y un exponente de clave pública, sino que generan solamente un módulo, de modo que el exponente le debe ser suministrado. Este exponente, para ser válido (como ay se dijo en el apartado 2.3 RSA), debe ser primo relativo con el módulo. La mejor forma de garantizar que dos número son primos

relativos (su máximo común divisor ha de ser 1) es elegir uno de ellos primo. Esto garantiza que serán primos relativos entre sí. Por tanto, elegimos un valor primo para el exponente público, `0x01, 0x00, 0x01`, que traducido del hexadecimal al decimal es 65.537, que es un exponente tremendamente usado en las claves públicas.

Para el caso de la clave privada, se expresan atributos como los ya mencionados en la función de crear una clave privada a partir de datos importados desde el exterior, destacando que se especifica que ha de ser privada `CKA_PRIVATE` con valor `CK_TRUE` y que es sensible con `CKA_SENSITIVE` también a `CK_TRUE`.

5.2.2.24 – Firmar Ficheros

```
CK_RV firmaFichero (FILE *datos, FILE *firmado, CK_OBJECT_HANDLE clavePrivda, CK_MECHANISM mecanismo);
```

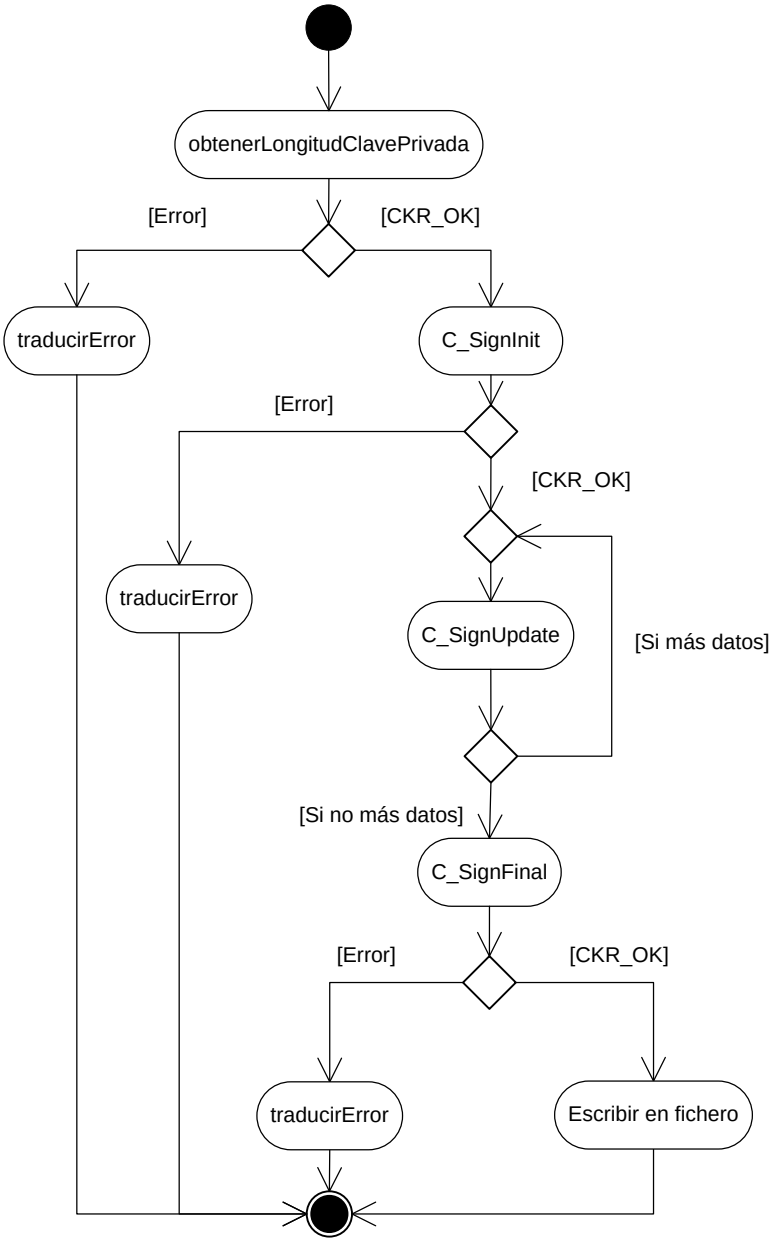

Ilustración 46: firmaFichero

El proceso de firma de un fichero comienza obteniendo la longitud de la clave privada que ha sido suministrada para realizar la firma, ya que el tamaño de la firma es el mismo que el tamaño del módulo de la clave privada (compartido con la pública). A continuación, se inicia el proceso de firmado con la invocación al Cryptoki `C_SignInit (sesion, &mecanismo, clavePrivda);` donde se expresa el mecanismo de firma que se va a utilizar. A continuación, se van actualizando los datos de la firma en un buffer interno realizando sucesivas llamadas a `C_SignUpdate (sesion, buff, lectura);` a la cual, en cada iteración, se le suministra un nuevo fragmento de datos a añadir a la firma. Estos datos están apuntados por `buff`. Una vez se ha concluido de firmar, todos los datos se obtiene el valor de la firma con la invocación a `C_SignFinal (sesion, firma, &longitud);` obteniendo en

`firma` el puntero a la firma digital. Después, se procede a la escritura en fichero de dichos datos.

5.2.2.25 – Firmar Buffers

```
CK_RV firmaBuffer (unsigned char *buffer, int lBuff, unsigned char *sign, CK_OBJECT_HANDLE clavePrivda, CK_MECHANISM mecanismo);
```

La función de firmar buffers de datos funciona exactamente igual que la de firmar a fichero. La única diferencia es que la firma no se escribe en un fichero, sino en una región de memoria previamente reservada que se le suministra a la función en la invocación (apuntada por `buffer`).

5.2.2.26 – Comprobar Firma de Ficheros

```
CK_RV comprobacionFirmaFichero (FILE *datos, FILE *firmado, CK_OBJECT_HANDLE clavePublica, CK_MECHANISM mecanismo);
```

El proceso de comprobación de un firma es igual al de firmar, incluido su esquema de actividad UML. La diferencia es en las funciones que son invocadas. En primer lugar, para comenzar la verificación, se invoca a `C_VerifyInit (sesion, &mecanismo, clavePublica);` donde se expresa el mecanismo a usar y el manejador de la clave pública. Luego, se invoca sucesivamente a `C_VerifyUpdate (sesion, buff, lectura);` donde se van pasando nuevos fragmentos del buffer apuntados por buff. Al final del proceso, se invoca a `C_VerifyFinal (sesion, firma, sizeof (firma));`. El retorno de la función será `CKR_OK` si se ha verificado al firma y otro valor en caso de no verificarse (el más típico de los errores es `CKR_SIGNATURE_INVALID` que significa que la firma no concuerda).

5.2.2.27 – Listar Mecanismos

```
CK_RV mecanismos (CK_BBOOL mostrar);
```

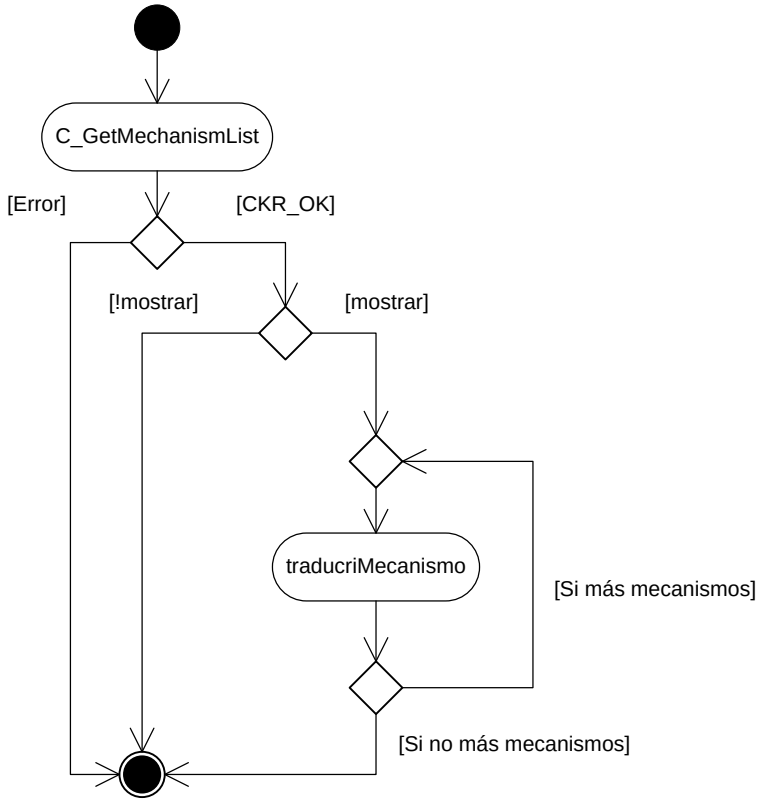

Ilustración 47: mecanismos

Esta función, en primer lugar, obtiene un listado de los mecanismos soportados por la tarjeta (más concretamente por el Cryptoki), invocando `C_GetMechanismList (lector, listaMecanismos, &contador);`. Como se ve, se hace uso de `listaMecanismos`, que es un puntero a la variable global, que contiene el espacio reservado para almacenar la lista de mecanismos. Una vez que se ha rellenado la lista con la consulta al Cryptoki, se puede discriminar entre mostrar resultados o no, en función del valor del argumento mostrar. Si se decide mostrar, se mira elemento a elemento de la lista de mecanismos y se invoca a la función `traducirMecanismo` para obtener una descripción textual de la misma y mostrarla por pantalla.

5.2.2.28 – Listar Mecanismos

```
CK_RV valorClavePublica (CK_OBJECT_HANDLE manejador,
char **valor);
```

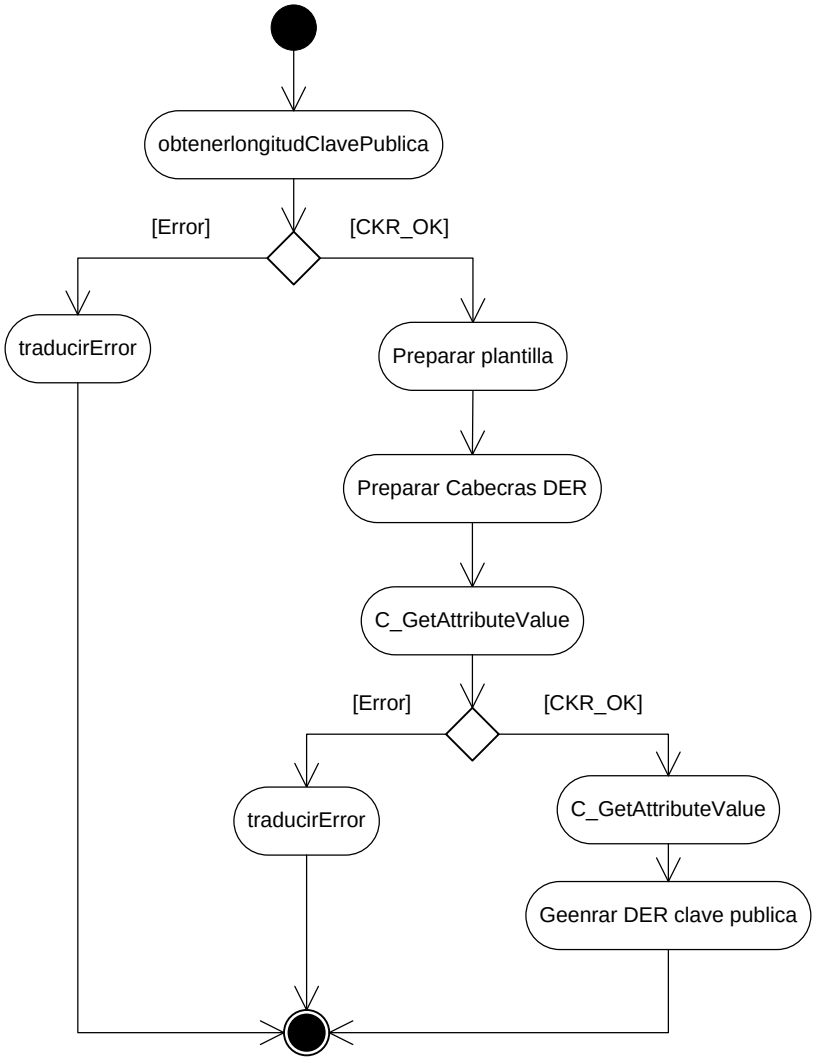

Ilustración 48: valorClavePublica

Esta función se encarga de obtener una codificación DER de la clave pública contenida dentro de una SmartCard. Esto se usa para poder incluir dicha clave dentro de un certificado, ya que la codificación es ligeramente diferente, entre clave pública de la tarjeta y la que va en el certificado (CSR para ser más exactos).

En primer lugar, se obtiene la longitud de la clave que es necesaria para poder generar la estructura DER de la clave propiamente dicha. Después, se preparan las cabeceras DER (tal y como se definió en el punto 2.1.1 ASN.1). A continuación, se prepara al plantilla de consulta para poder obtener el módulo y el exponente de la clave pública (son los dos únicos atributos necesarios). La plantilla usada es:

```
CK_ATTRIBUTE plantillaCP [] =
{
     {CKA_MODULUS, NULL_PTR, 0},
     {CKA_PUBLIC_EXPONENT, NULL_PTR, 0}
};
```

Con una primera invocación a `C_GetAttributeValue (sesion, manejador, plantillaCP, elementos);` obtenemos la longitud de los elementos a recuperar y, en la segunda invocación (previamente se ha reservado la memoria), se obtienen los valores del módulo y el exponente.

Cuando se tienen los valores se genera la estructura DER con esos valores. Finalmente, es devuelto ese buffer de memoria con la clave pública en formato DER que es posible utilizar para generar una CSR.

5.2.2.29 – Longitud de Certificado

```
CK_ULONG longitudCertificado (CK_OBJECT_HANDLE o);
```

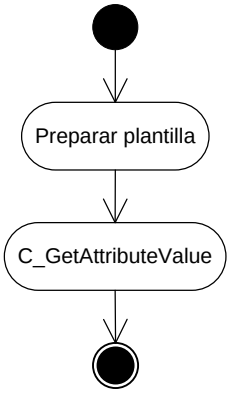

Ilustración 49: longitudCertificado

Obtener la longitud de un certificado es muy simple. En primer lugar, se genera una plantilla que permita obtener la longitud del atributo `CKA_VALUE`, que es el que contiene la codificación DER del certificado. La plantilla es:

```
CK_ATTRIBUTE certificateTemplate [] = {
     {CKA_VALUE, NULL_PTR, 0}
};
```

Con una llamada a `C_GetAttributeValue (sesion, o, certificateTemplate, NUM_ELEMTOS (certificateTemplate));` se obtiene la longitud del campo requerido en la plantilla y, como sólo se pretende obtener esa longitud, se retorna dicho valor y no es necesario hacer nada más.

Esta función se utiliza para obtener el tamaño de un certificado y poder hacer las reservas de memoria oportunas a la hora de exportar un certificado desde la tarjeta a fichero.

5.2.2.30 – Obtener Certificado

```
CK_RV obtenerCertificado (char *certificado, CK_ULONG lCert, CK_OBJECT_HANDLE manejador);
```

Esta función sigue un patrón idéntico al anterior. Como la longitud del certificado (para el parámetro CKA_VALUE) se le pasa por parámetro (es obtenido fuera de esta función) sólo es necesario realizar una llamada a `C_GetAttributeValue (sesion, manejador, plantillaCertificado, NUM_ELEMTOS (plantillaCertificado));` para obtener directamente el contenido de CKA_VALUE (que es la codificación DER del certificado). La plantilla que se usa para obtener el valor es similar a la de la función anterior:

```
CK_ATTRIBUTE plantillaCertificado [] =
{
    {CKA_VALUE, certificado, lCert}
};
```

5.2.2.31 – Mostrar Clave Pública

```
CK_RV mostrarClavePublica (CK_OBJECT_HANDLE manejador);
```

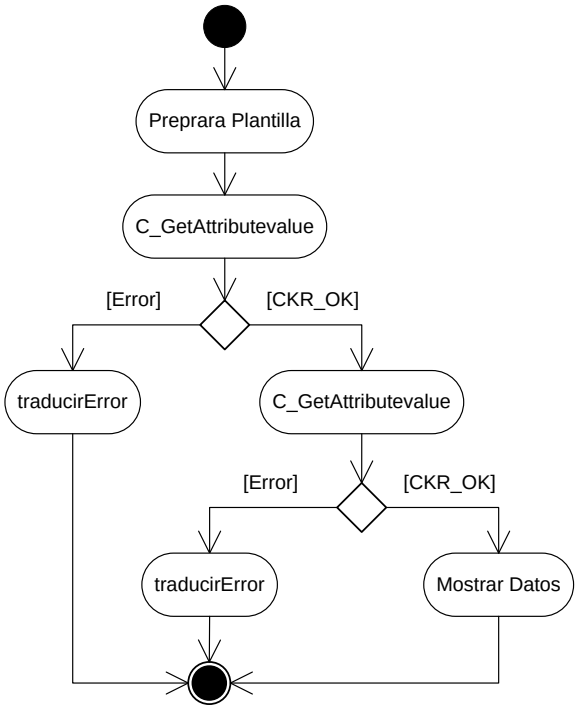

Ilustración 50: mostrarClavePublica

Esta función se encarga de mostrar la clave pública a partir de su manejador. Su funcionamiento es muy simple, primero se prepara la plantilla con los dos atributos que queremos recuperar, el módulo y el exponente:

```
CK_ATTRIBUTE plantillaCP [] =
{
      {CKA_MODULUS, NULL_PTR, 0},
      {CKA_PUBLIC_EXPONENT, NULL_PTR, 0}
};
```

Se realiza una primera invocación a `C_GetAttributeValue (sesion, manejador, plantillaCP, elementos);` para obtener las longitudes de los dos atributos (que son de tipo cadenas de bytes). Una vez se tienen las longitudes se realiza la pertinente reserva de memoria y se reinvoca a `C_GetAttributeValue (sesion, manejador, plantillaCP, elementos);` para obtener los valores de los argumentos de la plantilla. A continuación, se muestran los dos parámetros.

5.2.2.32 – Desbloquear PIN

```
CK_RV iniciarPIN (CK_BYTE_PTR PIN);
```

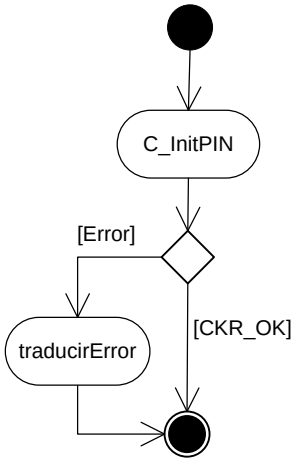

Ilustración 51: iniciarPIN

Esta función se encarga de desbloquear el PIN del usuario autenticado en el sistema. Se consigue invocando la funcionalidad del Cryptoki `C_InitPIN (sesion, PIN, strlen (PIN));` donde se le pasa el nuevo PIN y su longitud.

5.2.3 – Módulo LISTA

Este módulo es el encargado de gestionar la estructura de datos que almacenará la información necesaria para la gestión de los objetos de la tarjeta. La estructura que se encarga de esto es una lista enlazada de memoria dinámica.

Esta lista (sólo hay una dentro de la aplicación) es accedida a través de una variable global definida en el módulo PKCS#11 (`objeto *listaDeObjetos;`), aunque su definición, funcionalidades, etc. se encuentran en este módulo que se pasa a describir ahora.

En primer lugar, la lista no almacena toda la información disponible sobre cada objeto recuperado de la tarjeta, sólo almacena la clase a la que pertenece el objeto y su manejador. Con esta información se puede gestionar el objeto (a través de su manejador) y, sabiendo de qué tipo es (a través de su clase), se pueden aplicar las plantillas adecuadas.

De forma esquemática, la lista de objetos y cada nodo podrían ser así:

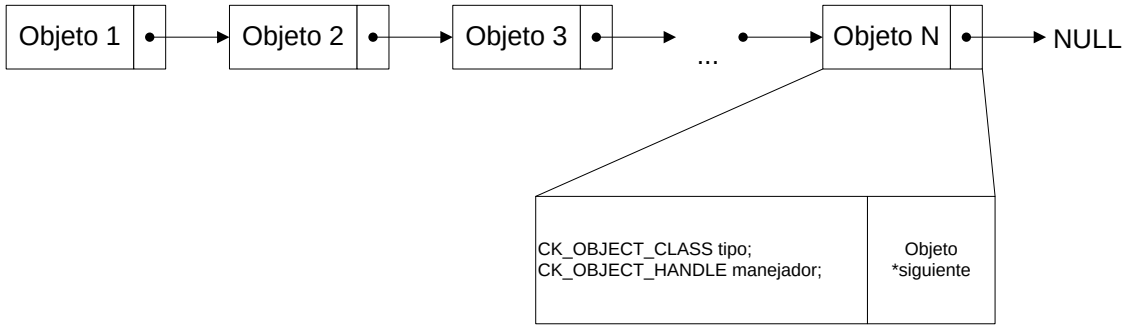

Ilustración 52: Lista de objetos

Cada vez que se produce un cambio de rol en la aplicación (login o logout), se elimina la lista y se consulta a la tarjeta nuevamente por los objetos disponibles, de modo que no se tenga acceso a algún manejador que no se pueda usar (como el de una clave privada cuando se es Usuario sin sesión). Esto no sería un problema, ya que el Cryptoki produciría un error al intentar gestionar objetos no accesibles, pero es mejor gestionar esto desde la aplicación para evitar esos errores.

Cada vez que se crea un nuevo objeto, éste es añadido a la lista con los datos que tienen en el momento de su creación sin necesidad de tener que consultar a la tarjeta por él. Cuando se eliminan objetos, se regenera la lista a fin de que no sean mostrados más.

El código que genera la lista está en listaObjetos.h y la funcionalidad está definida en listaObjetos.c. El código de la lista es el siguiente:

```
#ifndef _LISTA_
     typedef struct nodo objeto;
     struct nodo {
          CK_OBJECT_CLASS tipo;
          CK_OBJECT_HANDLE manejador;
          objeto *siguiente;
     };
     #define _LISTA_
#endif
```

En primer lugar, se observa que toda la estructura está incluida dentro de un bloque de preprocesado. Esto se hace así, ya que muchos módulos acceden a la lista y con esta estructura se consigue que no se produzcan redefiniciones. La primera vez que es incluido este código en compilación `_LISTA_` no está definida, por lo que se ejecuta todo el bloque definiendo el nodo de la lista. En el interior se define `_LISTA_` lo que supone que para próximos pases, `_LISTA_` ya estará definida y no se redefinirá el nodo (esto de no hacerse produciría errores de redefinición).

Las variables de tipo lista se definirán con el nombre objeto (definido en `typedef struct nodo objeto;`). Como se observa, el nodo tiene tres elementos, los dos primeros ya se han comentado, son el tipo de objeto y su manejador. El tercero es el puntero que señala el siguiente nodo de la lista. Cuando ese puntero vale `NULL` significa que es final de la lista y ya no hay más nodos.

De forma externa, a cada nodo de la lista se le asigna un identificador (un número entero comenzando desde 0) que es la forma de referenciar ese elemento de la lista. Estos identificadores pueden referenciar a toda la lista o sólo a ciertos tipos de nodos. Como se verá en la funcionalidad, la lista puede ser leída de inicio a fin mostrando todos los objetos o filtrando ciertos tipos. Tanto en un caso como en otro, el identificador representa el i-ésimo elemento de la lista (o sublista) al que se debe acceder para obtener los datos de su nodo.

A continuación, se definirán todas las funciones disponibles para este módulo LISTA. Debido a que en general son funciones genéricas de gestión de lista, sólo se hará una descripción de sus parámetros, los retornos y su funcionamiento, pero no se hará uso de diagramas UML.

5.2.3.1 Objeto Nuevo

```
objeto *nuevoObjeto ();
```

Esta función crea un nuevo nodo, pero no lo inserta en la lista. Reserva la memoria necesaria para albergar la información del nodo (vacío por defecto). El puntero a `siguiente` se establece como `NULL`, ya que por defecto todos los nuevos nodos que se inserten en la lista se añadirán en la última posición.

5.2.3.2 Crear Lista

```
void crearLista (objeto **lista);
```

Esta función inicializa una lista a `NULL`. Esta función está pensada para ser invocada con listas vacías, ya que, si se invoca sobre una lista (apuntada por `**lista`), se perderán los encales a los nodos haciendo imposible liberar posteriormente esa memoria.

5.2.3.3 Reinicializar plantilla

```
objeto *fin (objeto *lista);
```

Esta función devuelve un puntero al último nodo de la lista. Esto es necesario para poder insertar nuevos nodos, ya que, como se dijo, todos los nodos nuevos se insertarán al final de la lista y, para ello, es imprescindible conocer la dirección de memoria del último nodo. Si la lista está vacía, retornará `NULL`.

5.2.3.4 Reinicializar plantilla

```
void reiniciarPlantilla (CK_ATTRIBUTE plantillaTipo
[], CK_ULONG elementos);
```

Esta función se encarga de reiniciar las plantillas que se van a utilizar para gestionar objetos en las listas. Para reiniciar una plantilla, hay que recorrer todos sus elementos y asignar a `pValue NULL_PTR` y a `ulValueLen 0`. `plantillaTipo` es el puntero que referencia la plantilla a reinicializar y `elementos` es el número de entradas de dicha plantilla. Es muy importante tener en cuenta que la memoria

apuntada por `pValue` (en caso de ser dinámica y no una variable) debe liberarse por otros medios, ya que está función no libera memoria.

5.2.3.5 Recorrer Objetos

```
void recorrerObjetos (objeto *lista, CK_OBJECT_CLASS tipo);
```

Esta función recorre todos los elementos de la lista `lista` que coincidan con el tipo `tipo`. Los objetos serán mostrados por pantalla con los siguientes atributos:

- El tipo de objeto.
- La etiqueta del objeto.
- El ID del objeto (en hexadecimal).
- El número de serie (sólo para los certificados y en hexadecimal).

Acompañando a cada objeto, se mostrará (a su izquierda) su posición relativa dentro de la sublista. Esa referencia será la que se utilice para utilizar el objeto.

Cuando se quieren obtener todos los tipos de objetos, como en el caso de mostrar el contenido de la tarjeta, se debe pasar un `-1` como valor de `tipo`.

Hay que tener en cuenta que esta función invoca funcionalidad del Cryptoki concretamente `C_GetAttributeValue` para obtener los datos antes mencionados. Como todos los datos a recuperar, en general, son de longitud variable, serán necesarias dos invocaciones. En la primera, se obtendrá la longitud de los campos. Con esa longitud se hacen las reservas de memoria oportunas. En la segunda invocación, se recuperan los campos en sí.

Se deben usar dos plantillas diferentes: una con la etiqueta y el ID para los objetos en general y otra con esos atributos más el número de serie para los certificados (aunque no se hará así). Las plantillas deben ser inicializadas antes de invocar nuevamente a la función `C_GetAttributeValue` con un nuevo objeto o de lo contrario no se obtendrán resultados coherentes. En la aplicación se usa sólo una plantilla con los tres atributos. Esto se hace para ahorrar codificación, puesto que, cuando al Cryptoki se le solicita un valor de un atributo inexistente en un objeto concreto, lo que retorna es NULL_PTR.

Esta función recorre los elementos de la lista, de modo que sólo se mostrarán aquellos objetos que son visibles por el rol que solicita la invocación de la función, de manera que, si un usuario sin sesión obtiene el listado de objetos de la tarjeta, la lista no contendrá ninguna clave privada, de modo que no se mostrarán.

5.2.3.7 Elementos Tipo

```
int elementosTipo (CK_OBJECT_CLASS tipo, objeto *lista);
```

Esta función se encarga de contar el número de elementos de un tipo dado en el argumento `tipo` de la lista `lista`. La función recorre la lista desde el inicio al final incrementando un contador cada vez que encuentra un elemento del tipo dado. Esta función se usa frecuentemente en el módulo principal para evitar ejecutar funciones cuando no se cumplen los requisitos previos, como, por ejemplo, evitar ejecutar la función de firmar un fichero si no hay ninguna clave privada en la tarjeta (sin claves privadas no se puede firmar).

Como siempre el -1 representa el tipo genérico, lo que quiere decir que cuente todos los elementos.

5.2.3.8 Manejador Posición

```
CK_OBJECT_HANDLE manejadorPosicion (objeto *lista, int posicion, CK_OBJECT_CLASS tipo);
```

Esta función retorna el manejador del objeto i-ésimo referenciado por el índice `posicion`, de la lista `lista` y del tipo `tipo`. Esta función está muy ligada a los identificadores que se le muestran al usuario para que seleccione objetos. Ese identificador elegido por el usuario se usará como índice para buscar el nodo dentro de la lista. Una vez se ha localizado el nodo se retorna su manejador para que pueda ser usado el objeto.

5.2.3.9 Insertar

```
void insertar (CK_OBJECT_CLASS tipo, CK_OBJECT_HANDLE manejador, objeto **lista);
```

Esta función inserta el objeto de tipo `tipo` cuyo manejador es `manejador` en la lista `lista`. Para ello, en primer lugar, reserva memoria para construir el nodo invocando `objeto o = nuevoObjeto ();` y asignando la dirección en el puntero o. Luego, busca en la lista el último elemento e inserta el objeto en la última posición.

5.2.3.10 Borrar

```
void borrar (objeto **lista);
```

Esta función elimina todos los objetos de la lista apuntada por `lista`, dejándola con valor `NULL`. Al contrario que otras funciones mencionadas, esta función sí que libera la memoria de la lista. El algoritmo es simple. Se busca el último nodo de la lista, se libera la memoria, se hace que `siguiente` apunte a `NULL` y se repite el proceso.

5.2.4 – Módulo MECANISMOS

Este es un módulo auxiliar que provee de ciertas funcionalidades relacionadas con los mecanismos. En general, este módulo ofrece soporte al uso de ciertos mecanismos, ya que se encarga de decir qué mecanismos pueden ser utilizados bajo ciertos tipos de operaciones, como, por ejemplo, qué mecanismos se pueden utilizar para firmar.

Este módulo está compuesto por los ficheros mecanismos.c donde están las implementaciones de los algoritmos y mecanismos.h donde están las definiciones.

El módulo citado sólo provee de tres funciones (relacionadas con uso de mecanismos y tamaños de cadenas de mecanismos) que se describen en los siguientes puntos:

5.2.4.1 Tamaño Mecanismos

```
CK_RV tamMecanismos (CK_SLOT_ID lector, CK_ULONG_PTR
tam);
```

Esta función permite obtener el tamaño de la lista de mecanismos presentes en la tarjeta insertada en el `lector` referenciado por lector. La longitud se devolverá en la dirección de memoria apuntada por `tam`. Puesto que los mecanismos son `CK_MECHANISM_TYPE` que al final es sinónimo de `unsigned long int`, su tamaño (en máquinas de 32 bits) es de 4 bytes, por lo que la longitud devuelta en la dirección de `tam` será el número de mecanismos de la tarjeta multiplicado por 4 bytes de cada mecanismo.

5.2.4.2 Tamaño Compatibles

```
CK_ULONG tamCompatibles (CK_SLOT_ID lector, CK_MECHANISM_TYPE_PTR mecanismo, CK_ULONG tam, CK_FLAGS busqueda);
```

Esta función retorna el tamaño de los mecanismos compatibles con un tipo de operación concreta a realizar. Los mecanismos se obtienen del array apuntado por `mecanismo`. El tamaño total del array es `tam` y el tipo de operación a realizar se expresa en `busqueda`. Esta función es utilizada para obtener el tamaño total de los mecanismos presentes en la tarjeta que sirven para una operación concreta (generar claves, por ejemplo).

El algoritmo se basa en obtener la información del mecanismo con `C_GetMechanismInfo (lector, mecanismo [i], &mecInfo);` donde `mecanismo [i]` es el mecanismo del que se desea obtener la información y `mecInfo` una estructura de tipo `CK_MECHANISM_INFO` donde se almacenará la información. La información no es más que una región de memoria de 32 bits, donde cada posición es un flag que representa una acción (por tanto, hay un máximo de 32 acciones representables).

5.2.4.3 Tamaño Compatibles

```
void obtenerArrayCompatibilidad (CK_SLOT_ID lector, CK_FLAGS busqueda, CK_MECHANISM_TYPE_PTR original, CK_ULONG tamOri, CK_MECHANISM_TYPE_PTR lista);
```

Esta función se utiliza para obtener una sublista obtenida del array global de mecanismos referenciado por `original`. Los mecanismos que concuerdan en funcionalidad con lo expresado en búsqueda son añadidos al array apuntado por lista (la memoria debe haber sido reservada previamente). Al finalizar la función, se tiene un array con todos los manejadores de los mecanismos que son compatibles con la acción expresada. El funcionamiento es simple: se obtienen los 32 bits de flags del mecanismo actual (se recorren todos de inicio a fin), se aplica un AND lógico con el flag `busqueda` pasado en la función y, si el resultado es igual al flag, significa que el bit del flag estaba presente en la información del mecanismo y, por tanto, el mecanismo concuerda con la funcionalidad buscada. Si se cumple la operación lógica, se copia esa posición del array `original` en la posición siguiente del array `lista`.

5.2.5 – Módulo SOPORTE

Este módulo se encarga de englobar cualquier función de soporte que deba ser utilizada en cualquiera de los otros módulos (preferiblemente aquellas que son requeridas por más de un módulo): funciones de traducción de mensajes, de mantenimiento de la pantalla de la consola, de conversiones y comparación de cadenas, etc. Cualquier funcionalidad auxiliar, en general, que sirva para realizar un algoritmo, pero que sólo sirva para darle soporte, estará incluida en este módulo.

Soporte se compone de los ficheros soporte.c y soprote.h. En soporte.h. Además de las definiciones de las funciones, se describen dos macros de la aplicación que serán frecuentemente utilizadas por el resto de los módulos:

```
#define NUM_ELEMTOS(plantilla) (sizeof (plantilla) / sizeof (plantilla [0]))
```

Esta macro cuenta el número de posiciones del array apuntado por `plantilla`, realizando la operación de dividir la `plantilla` entre el tamaño de su primera posición (tamaño total entre tamaño de la primera posición es igual al número de posiciones).

```
#define LONG_ELEM(plantilla, i) (plantilla [i].ulValueLen)
```

Esta macro devuelve el tamaño del atributo de plantilla de la posición `i`. La funcionalidad es simple. Simplemente retorna `plantilla [i].ulValueLen` que es la longitud del atributo i-ésimo de la plantilla.

Las funcionalidades implementadas en este módulo se describirán en los siguientes apartados.

5.2.5.1 Mostrar Campo

```
void mostrarCampo (char * valor, int longitud, int salto, int esp);
```

Esta función muestra por pantalla, en formato ASCII, la cadena apuntada por `valor` donde longitud es la `longitud` en bytes de dicha cadena. Si `salto` es 1, se hace un salto de línea al terminar. `esp` marca el número de espacios en blanco que deben dejarse antes de escribir la cadena valor.

5.2.5.2 Mostrar Campo Hexadecimal

```
void mostrarCampoHex (char * valor, int longitud, int salto, int esp);
```

Esta función sigue un algoritmo exactamente igual que la anterior, sólo que `valor` no es mostrada en formato ASCII, sino en codificación hexadecimal separando cada bytes por ":" puntos (1 byte se corresponde con dos caracteres hexadecimales).

5.2.5.3 Mostrar Clase

```
void mostrarClase (CK_OBJECT_CLASS clase);
```

Esta función muestra una traducción a ASCII de la clase referenciada en `clase`. Un switch se encarga de imprimir una cadena u otra en función del valor de `clase` (que es de tipo entero).

5.2.5.4 ¿Es Entero?

```
CK_BBOOL esEntero (char *entrada);
```

Esta función analiza los bytes que componen la cadena `entrada`. En caso de encontrar un byte con valor mayor que 57 o menor que 48 (menor que "0" o mayor que "9"), retornará `CK_FALSE`. Si todos los bytes que la componen son números ASCII, entonces retornará `CK_TRUE`.

5.2.5.5 Valor del parámetro

```
CK_BBOOL paramValor (char *entrada, int *num, int min, int max);
```

Esta función retornará `CK_TRUE` si el parámetro `entrada` es un número y `CK_FALSE` si no lo es. En caso de ser un número, el valor de dicho número es retornado en la dirección de memoria apuntada por `num`. `min` y `max` son los límites del número que puede contener `entrada`. En caso de que `entrada` sea un número válido, pero su traducción a entero no esté en el rango `[min, max]` la función también retornará `CK_FALSE`.

5.2.5.6 Comparar Cadenas

```
CK_BBOOL compararCadenas (CK_BYTE_PTR cad1, CK_BYTE_PTR cad2, int longitud);
```

Esta función retornará `CK_TRUE` si todos los bytes de `cad1` coinciden con los bytes de `cad2` y, además, las longitudes de ambas son iguales. En caso de que algo no case, retorna `CK_FALSE`.

5.2.5.7 Traducir Mecanismos

```
char *traducirMecanismos (CK_MECHANISM_TYPE mecanismo);
```

Esta función retorna una cadena de texto describiendo el mecanismo referenciando en `mecanismo`.

5.2.5.8 Traducir Error

```
char *traducirError (CK_RV error);
```

Esta función retorna una cadena de texto describiendo el error expresado en `error`.

5.2.5.9 Obtener Longitud

```
int obtenerLongitudReal (char * original, int longitud);
```

Esta función se usa para obtener longitudes de ciertos buffers que pueden contener el carácter /0 (fin de cadena) como podrían ser módulos o exponentes de claves públicas o privadas. `longitud` es una longitud de un buffer de memoria.

5.2.5.10 Invertir Bytes de Cadena

```
void invertirBytesCadena (char *original, int lDestino, char *destino);
```

Esta función recibe una cadena origen referenciada por `original` y su longitud está expresada en `lDestino`. Lo que hace esta función es insertar la cadena `origen` en orden inverso en la cadena apuntada por `destino`.

5.2.5.11 Asignar Entero a Cadena

```
void asignarEnteroACadena (int num, CK_CHAR fecha [], int len);
```

Esta función introduce el número `num` en el array fecha. El número ocupará los caracteres más a la derecha de la cadena, de modo que el resto de la cadena es rellenada con 0 (ASCII 48).

5.2.5.12 Formato Fecha

```
CK_DATE formatoFecha (ASN1_UTCTIME *tm, CK_BBOOL *correcto);
```

Esta función convierte un UTC TIME (tipo de datos de OpenSSL que referencia las fechas de los certificados) en una fecha de tipo CK_DATE (estructura compuesta de enteros para referencias día, mes y año). El algoritmo es una copia literal del usado por OpenSSL para mostrar en formato ASCII entendible las cadenas contendías en los certificados con las fechas (UTC TIME). Si la fecha es correcta, `correcto` valdrá `CK_TRUE` y, en caso contrario, `CK_FALSE` (caso de mes 13 por ejemplo).

5.2.5.13 Pausa

```
void pausa ();
```

Esta función espera una pulsación de entre para terminar. Se usa un `scanf` para leer un carácter (vaciando previamente el buffer de entrada `stdin`).

5.2.5.14 Limpiar

```
void limpiar ();
```

Esta función limpia la consola. En Windows se hace con una invocación al comando `cls` con `system ("cls");`

5.2.5.15 Corregir Cadena

```
void corregirCadena (unsigned char *cadena, int longitud);
```

Esta función corrige cadenas de entrada, eliminando caracteres problemáticos, como podrían ser: letras con tilde, la ñ o caracteres del castellano que no están incluidos en ASCII de 7 bits. La cadena referenciada por `cadena` es recorrida de inicio a fin, y los caracteres "problemáticos" son sustituidos por espacios (ASCII 32).

5.2.5.16 Mostrar Lista de Lectores

```
void mostrarListaLectores (CK_SLOT_ID_PTR lista, CK_ULONG tam);
```

Esta función muestra los lectores contenidos en `lista` (el array de todos los lectores presentes en el sistema). Por definición, lista sólo contiene lectores con un token presente, de modo que no se mostrarán, en este listado, aquellos electores conectados al sistema, pero que no tienen un SmartCard insertada.

La información del lector se muestra invocando la función `infoSlot (lista [i]);` definida en el módulo PKCS#11.

5.2.6 Módulo OPENSSL

Este es el módulo de la aplicación que se encarga de invocar e interactuar con la API de OpenSSL. Se ha llamado del mismo modo, pero no son lo mismo: por un lado, está la API expresada por la DLL libeay32.dll y sus correspondientes ficheros de cabecera .h y, por otro, este módulo, OPENSSL expresado por dos ficheros openssl.c y

openssl.h, que son los que describen la interacción entre dicha DLL y la aplicación desarrollada en este proyecto.

En ese módulo, se trabaja principalmente con dos PKCS, el 10 y el 12, además de con certificados digitales. Con PKCS#10 se permite la generación de CSR en la aplicación. Esto se usa para poder crear peticiones de certificados a partir de las claves generadas en la tarjeta. Se van a tratar dos tipos de CSR: el primero, enfocado a obtener certificados genéricos para firma digital y, el segundo, CSR, para obtener certificados de logon en sistemas Windows.

Con la interacción de PKCS#12, se consigue entender ficheros PFX y poder importar los perfiles digitales que éstos contienen (claves pública y privada y certificado digital).

Este módulo también se encarga de la gestión de certificados X.509, ya sea: para importarlos a la tarjeta, exportarlos a un fichero o mostrarlos por pantalla.

Todas estas operaciones se realizan a través de la API de OpenSSL. Es una API enorme, y ha sido necesario, en muchas ocasiones, ver el código fuente del propio proyecto OpenSSL para poder entender su uso o funcionamiento, ya que no toda la API esta comentada. En ocasiones, también ha sido necesario ver cómo se realizan funcionalidades en OpenSSL para poder imitarlas en este proyecto puesto que OpenSSL usa multitud de pequeñas funciones más a bajo nivel que las comentadas en la API y que son inaccesibles sin ver el código, como es el caso de la firma de una CSR con la clave privada, ya que no hay ninguna funcionalidad de API como tal que permite hacer esto.

Todas las interacciones entre la aplicación y el disco (ficheros), a nivel de este módulo, se realizan mediante objetos BIO, que son objetos que proveen una funcionalidad abstracta de entrada/salida. OpenSSL tiene la posibilidad de trabajar con ficheros genéricos, pero esta opción dio problemas en la implementación; pues, en ciertos sistemas operativos; producía errores (distintas versiones de Windows).

Para tratar la definición de las funcionalidades, no se usarán diagramas UML de actividad, puesto que son funciones bastante lineales, donde la complejidad no radica en el algoritmo en sí, sino: en las invocaciones concretas, su orden y los argumentos con los que se realizan. Por tanto, lo que sí que se hará es presentar en muchas ocasiones fragmentos de código que reflejen lo que se está haciendo.

A continuación, se pasa a describir las funciones que componen este módulo.

5.2.6.1 Crear Petición de Certificado

```
void crearReqCert (char *clave, CK_ULONG lClave);
```

En primer lugar, se definen las matrices que contendrán los datos asociados al CSR: una matriz es para los datos de CSR de firma y otra para los de logon. Las matrices tienen dos columnas: en la primera, se expresa el identificador y, en la segunda, el valor asociado a dicho identificador. Las matrices son las siguientes:

```
struct entrada entradasSTD [DATOSPERSONALES] =
{
     {"countryName", ""},
     {"stateOrProvinceName", ""},
     {"localityName", ""},
     {"organizationName", ""},
     {"organizationalUnitName", ""},
     {"commonName", ""},
     {"emailAddress", ""}

};
    struct entrada entradasLO [DATOSLOGON] =
{
     {"commonName", ""},
     {"commonName", ""},
     {"domainComponent", ""},
     {"domainComponent", ""}
};
```

Lo que se hace, a continuación, es solicitarle al usuario los datos para rellenar esos campos (información personal). Luego, se usa el objeto `BIO mem` para almacenar la clave pública que se pasó por parámetro con `BIO_write (mem, clave, lClave);` y se convierte la clave de codificación DER a PEM con `pub = d2i_PUBKEY_bio (mem, NULL);`. Luego, se crea el objeto CSR con `req = X509_REQ_new ();` y se le añaden los datos personales de las matrices anteriores a una estructura de tipo `X509_NAME`, que será incorporada a `req`. En este momento, se le añade la clave pública a la petición con `X509_REQ_set_pubkey (req, pub);`.

Después, es momento de añadir los usos de la CSR que serán los futuros usos del certificado X.509. Para el caso de CSR, para firma digital se usará:

```
X509_EXTENSION *ext;
STACK_OF (X509_EXTENSION) *extlist;
extlist = sk_X509_EXTENSION_new_null ();
```

```
ext = X509V3_EXT_conf (NULL, NULL, "keyUsage",
"digitalSignature");
sk_X509_EXTENSION_push (extlist, ext);
ext = X509V3_EXT_conf (NULL, NULL, "keyUsage",
"nonRepudiation");
sk_X509_EXTENSION_push (extlist, ext);
ext = X509V3_EXT_conf (NULL, NULL, "keyUsage",
"keyEncipherment");
```

donde se está expresando que el uso será `digitalSignature`, `nonRepudiation` y `keyEncipherment` (firma digital, no repudio y cifrado de claves). En el caso de logon se usaría:

```
sprintf (UPN,
"otherName:1.3.6.1.4.1.311.20.2.3;UTF8:%s@%s.%s", sel
[1].value, sel [3].value, sel [2].value);
ext = X509V3_EXT_conf (NULL, NULL, "subjectAltName", UPN);
sk_X509_EXTENSION_push (extlist, ext);
ext = X509V3_EXT_conf (NULL, NULL, "extendedKeyUsage",
"1.3.6.1.4.1.311.20.2.2");
sk_X509_EXTENSION_push (extlist, ext);
ext = X509V3_EXT_conf (NULL, NULL, "extendedKeyUsage",
"1.3.1.5.5.7.3.2");
sk_X509_EXTENSION_push (extlist, ext);
```

Donde, en primer lugar, se genera el UPN del usuario en formato nombre@dominio.extensión. Y, luego, se añaden dos usos `1.3.6.1.4.1.311.20.2.2` y `1.3.1.5.5.7.3.2` (Inicio de sesión de tarjeta inteligente y Autenticación del cliente).

Una vez se han añadido los usos a la CSR, ésta se firma con la clave privada del usuario (la que se haya seleccionado para tal fin). Esto se hace con `firmaBuffer (buf, lbuf, sign, clavePrivda, mecanismo);` (función definida en PKCS#11).

Al final, se escribe, en un fichero, la petición ya firmada usando objetos BIO. Esto se hace del siguiente modo:

```
pmp = BIO_new_file (fichero, "w");
PEM_write_bio_X509_REQ (pmp, req);
```

5.2.6.2 Leer Certificado

```
void leerCert (CK_OBJECT_HANDLE clave);
```

Esta función lee un certificado de un fichero y la asocia a la clave privada seleccionada de la tarjeta. Para ello, en primer lugar, se lee el certificado desde el fichero usando objetos `BIO` con:

```
BIO *pmp = BIO_new (BIO_s_file());
pmp = BIO_new_file (entrada, "r");
```

donde entrada es el nombre y ruta del fichero que contiene el certificado. Luego, se obtiene el objeto `X509` desde el objeto BIO del siguiente modo `cert = PEM_read_bio_X509 (pmp, NULL, NULL, NULL);`. Una vez tenemos el certificado en `cert`, podemos transformarlo de PEM a DER para poder introducirlo en la tarjeta. Esto se realiza con la llamada `lbuf = ASN1_item_i2d ((void *) cert, &buf, ASN1_ITEM_rptr (X509));`. Finalmente, se invoca la función de PKCS#11, encargada de crear certificados en la tarjeta con la siguiente llamada:

```
crearCertificado (etiqueta, buf, lbuf, cert->cert_info->
serialNumber->data,
cert->cert_info->serialNumber->length, cert->cert_info->
issuer->bytes->data,
cert->cert_info->issuer->bytes->length, cert->cert_info->
subject->bytes->data,
cert->cert_info->subject->bytes->length, fechaNB, fechaNA,
(time_t)getTime);
```

5.2.6.3 Leer PKCS#12

```
void leerPKCS12 ();
```

En primer lugar, se preparan los objetos básicos que van a ser usados en la lectura del PFX

```
PKCS12 *pkcs12 = NULL;
BIO *pmp = BIO_new (BIO_s_file());
X509 *cert = NULL;
EVP_PKEY *privateKey = NULL;
```

El objeto `pkcs12` es el objeto que tratará lo obtenido del fichero PFX. `pmp` es el objeto BIO que se encargará de leer el PFX. `cert` será el objeto que contendrá el certificado y `privateKey` la clave privada. No hay objeto para la clave pública,

puesto que, como la privada contiene el módulo y el exponente públicos que componen la clave pública, no será necesario crear un objeto para la clave pública en sí.

El funcionamiento de la función sería el siguiente: en primer lugar, se obtiene el contenido del fichero PFX, y se almacena en el objeto `pkcs12`. Esto se hace con las invocaciones: `BIO_read_filename (pmp, entrada);` y `pkcs12 = d2i_PKCS12_bio (pmp, NULL);`: la primera lee el fichero y la segunda obtiene un objeto PKCS#12 en formato DER.

Los ficheros PFX suelen estar encriptados con una clave simétrica (para proteger la clave privada que contienen). Por tanto, para poder extraer los objetos, anteriormente descritos, desde el PFX mediante el `password` correcto, se usa la invocación `PKCS12_parse (pkcs12, password, &privateKey, &cert, NULL);`.

Una vez se tienen los objetos codificados en DER, se invocan las funciones del módulo PKCS#11 `crearClavePublica`, `crearClavePrivada` y `crearCertificado` para insertar los objetos obtenidos en la tarjeta. El caso concreto de la clave pública difiere de unas tarjetas a otras. Concretamente, las tarjetas Ceres, al crear un objeto clave privada, dentro de la tarjeta, inmediatamente generan el objeto clave pública, de modo que, si se invocara la creación de la clave pública, se produciría un error. Para evitar esto, se hace uso del preprocesador del siguiente modo:

```
#ifndef CERES
     rv = crearClavePublica (…);
      if (rv == CKR_OK)
          printf ("… creado correctamente...\n");
#endif
```

5.2.6.4 Exportar Certificado X.509

```
void exportarX509 ();
```

Esta función es relativamente sencilla (en comparación con las anteriores de este mismo módulo). Usa: un objeto de tipo `X509` para poder escribir el certificado a fichero, un objeto `BIO` para poder escribirlo y un manejador de objeto de la tarjeta para poder obtener el valor del certificado de la tarjeta. La definición de los mismos sería la siguiente:

```
X509 *cert = NULL;
```

```
CK_OBJECT_HANDLE certificado;
BIO *mem = BIO_new (BIO_s_mem());
```

En primer lugar, se obtiene el certificado de la tarjeta a partir de su manejador con las siguientes invocaciones: `lCert = longitudCertificado (certificado);` que nos devuelve la longitud del certificado y `obtenerCertificado (valor, lCert, certificado);` con el que obtenemos un buffer con la codificación DER del certificado.

Una vez tenemos el buffer con la codificación DER, creamos el objeto certificado invocando `cert = d2i_X509_bio (mem, NULL);` y lo escribimos en el fichero con `BIO_write (mem, valor, lCert);`.

5.2.6.5 Mostrar Certificado X.509

```
void mostrarX509 ();
```

Esta función tiene un comportamiento exactamente igual que la anterior, con la única diferencia de que, una vez se ha obtenido el certificado, no se escribe en un fichero, sino que se invoca a la API para mostrarlo por pantalla (tal y como hace la aplicación openssl).

Una vez se tiene el objeto `X509` con el certificado, se invoca a `X509_print_fp(stdout,cert);` para mostrarlo por la salida estándar.

6 – CONCLUSIONES

En primer lugar, este proyecto ha servido para adquirir unos conocimientos sobre el estándar PKCS#11 de gestión de SmartCards (enfocadas a la criptografía). Gracias a él, se ha comprendido el funcionamiento de las tarjeas criptográficas: cómo están configuradas, sus capacidades, cómo se interactúa y todo lo relacionado, en general, con el manejo y uso de las SmartCards.

Estos conocimientos no se limitan al campo teórico, ya que se trata de unos conocimientos con una orientación eminentemente práctica, de modo que, gracias a ellos, se podría realizar cualquier proyecto donde el objetivo fuera trabajar con tarjetas criptográficas.

PKCS#11 es un estándar tremendamente amplio que da soporte a la práctica totalidad de la criptografía simétrica y asimétrica, sumas de verificación y aspectos relacionados. Aunque este proyecto en su parte de implementación se haya centrado en las firmas digitales (campo más importante de la criptografía de clave pública o asimétrica), el estudio de este estándar he permitido adquirir capacidades para utilizarlo de forma íntegra (incluidas aquellas partes que no se han trabajado a nivel de desarrollo en el presente proyecto). De hecho, hubo funcionalidad implementada que, finalmente, no se incluyó al proyecto por estar fuera de sus límites como fue la gestión de las sumas de verificación SHA1 y MD5.

El estudio de las necesidades del sistema (en su fase de análisis) llevó a la conclusión de la importancia de utilizar OpenSSL para la gestión de todos los aspectos PKI del proyecto, tales como gestión de certificados, ficheros PFX, etc. Para poder gestionar la API de OpenSSL (ya que era el único método factible de usar el proyecto OpenSSL), fue necesario estudiarla y entenderla. Esto supuso que se adquirieron una serie de destrezas para gestionar arquitecturas PKI con OpenSSL necesarias para la finalización del proyecto, aunque, inicialmente, no se contempló como un objetivo el dominio de dicho proyecto.

El primer objetivo que se planteó al inicio del proyecto fue la comprensión del estándar PKCS#11 y, tal como se ha mencionado en estas conclusiones, es algo que se ha conseguido. La prueba palpable de dicha consecución es la implementación de la aplicación de este proyecto, capaz de gestionar tarjetas criptográficas mediante el uso de PKCS#11.

El siguiente objetivo venía marcado por la necesidad de obtener códigos que fueran portables entre plataformas. El lenguaje que ofrece más portabilidad debido a su máquina virtual es Java, pero, tal y como se explicó en el análisis, era inviable su uso, por lo que hubo que recurrir a C para el desarrollo de la aplicación. Para garantizar

que el código fuera portable entre distintos sistemas operativos, fue necesario hacer la implementación en ANSI C, evitando utilizar cualquier librería o biblioteca no estándar. Esto también se consiguió, ya que todo lo necesario para la implementación puede ser gestionado con ANSI C (no siempre de la forma más fácil y rápida, pero siempre es factible de hacerse). El uso de otros proyectos como OpenSSL sigue garantizando la portabilidad, ya que OpenSSL tiene versiones para Unix y Windows por lo que el código fuente que use la API OpenSSL podrá ser posteriormente compilado entra plataforma.

El presente proyecto ha sido realizado en Windows y sólo se ha compilado y desarrollado dicha versión sin hacerse una compilación para Linux. Esto se debe a que es necesario un Cryptoki (módulo de PKCS#11 para la tarjeta) desarrollado para un sistema concreto y, por desgracia, las tarjetas disponen siempre de su módulo para Windows, pero casi nunca para Linux, por lo que se hacía casi imposible gestionar las tarjetas bajo dicho sistema.

El siguiente objetivo que se planteó fue la gestión de los objetos de las tarjetas criptográficas. En este proyecto, a través del uso de PKCS#11, y apoyándose en OpenSSL para la gestión de elementos PKI, se ha logrado manejar, de forma total, los objetos de las SmartCards. Se permite su creación (ya sea interna o una importación desde un fichero), su eliminación, su modificación (al menos aquellos aspectos de los objetos que no afectan a su funcionamiento) y su visualización, en el caso de los certificados y las claves públicas, es posible ver por completo su contenido.

Otro objetivo era el poder incluir dentro de las SmartCards identidades digitales expresadas mediante el estándar PKCS#12. Esto es lo que se conoce como ficheros PFX (son identidades digitales porque contienen certificado y par de claves de una persona o entidad). El permitir la importación de estos PKCS#12 ofrece la posibilidad de crear claves y certificados por terceros bajo cualquier tipo de aplicación o plataforma y, posteriormente, insertarlos en la tarjeta donde podrán ser usados como si de objetos nativos se tratase. Gracias a la potencia de la API de OpenSSL, la tarea de gestionar estas importaciones ha sido bastante sencilla y solventada con total éxito. De hecho, se permite la importación de cualquier PFX esté en texto en claro o encriptado mediante el uso de cualquier tipo de algoritmo de cifrado simétrico (práctica usual y recomendada para estos ficheros PFX).

Para garantizar una mayor seguridad, era necesario poder generar los pares de claves (públicas y privadas) dentro de la propia tarjeta, sin necesidad de crearlas con un programa en un ordenador y, posteriormente, incluirlas a la tarjeta (con PKCS#12 por ejemplo). Mediante el uso de PKCS#11 se logró este objetivo y en este proyecto se ha conseguido generar los pares de claves RSA en el interior de la tarjeta con el uso exclusivo del hardware de la tarjeta. Además, para dotar a dichas claves de mayor seguridad, se ha permitido la generación de claves de 1024 bits o 2048, siempre que el

hardware de la tarjeta lo permita, ya que hay tarjetas que no permiten la generación o uso de claves de más de 1024 bits.

En este proyecto, para dotar de más potencialidad al mismo, era necesario poder hacer uso de las claves generadas más allá de la firma y su verificación (que es sin duda lo más importante). Se planteó, para ello, el objetivo de poder generar peticiones de certificados en formato PKCS#10, también denominadas CSR. Estas peticiones deberían ser de dos tipos: peticiones generales, enfocadas a obtener un certificado de uso general (firma digital) y peticiones enfocadas a obtener certificados válidos para hacer login en sistemas Windows con arquitectura PKI. Nuevamente, gracias al estudio y dominio de OpenSSL y su API, se consiguió integrar su funcionalidad con PKCS#11, de modo que se podían generar CSR y firmarlos con la clave privada almacenada en la tarjeta (a que toda CSR debe ir firmada para ser admitida por la CA). Ambos tipos de CSR fueron generadas satisfactoriamente. Para las generales, se utilizó una CA generada con la aplicación (no la API) openssl, pudiéndose firmar la petición sin problemas y obteniendo un certificado válido (comprobado su consistencia con Windows y con el navegador Mozilla Firefox). Para las de logon, se recurrió a una CA soportada por un Windows Server 2003 la cual, al igual que sucedió con openssl, firmó sin ningún problema la petición de logon, obteniendo un certificado con el cual se pudo hacer logon en el sistema mediante la tarjeta si ningún tipo de problema.

El último, y más importante objetivo, era el de la consecución de firmas digitales de ficheros y su posterior verificación. La firma digital es la máxima expresión de la criptografía pública ofreciendo no repudio, integridad e incluso, bajo ciertos esquemas, autenticación. Era, por tanto, vital poder utilizar claves RSA dentro de la tarjeta para firmar ficheros y poder comprobar, posteriormente, esas firmas. Mediante el uso nuevamente de PKCS#11 se logró realizar dicha tarea completamente, permitiendo la firma de ficheros con todos los mecanismos hardware soportados por cada tarjeta y, posteriormente, permitiendo su verificación

Para concluir, se puede decir que se han logrado de forma satisfactoria todos los objetivos marcados para el proyecto y también algunas cosas que no fueron planteadas como objetivos inicialmente (como es todo lo relacionado con OpenSSL). Además, ha servido para entender y aprender a usar el estándar PKCS#11 que es el más importante de todos los mecanismos de interacción con tarjetas criptográficas. En último lugar, el proyecto ha servido, también, para complementar la funcionalidad ofrecida por PKCS#11 enfocada a la gestión de las targets con la de OpenSSL, enfocada a la gestión de arquitecturas PKI ofreciendo todo un abanico de nuevas posibilidades de ampliación a futuro.

7 – FUTUROS DESARROLLOS

7.1 – IMPLEMENTACIÓN DE UNA GUI

Un apartado importante que debería ser desarrollado en futuras ampliaciones de este proyecto es la creación de una Interfaz Gráfica de Usuario. Es cierto que no se generó una debido a la necesidad de tener un proyecto portable que pudiera ser compilado sin apenas cambios tanto en Windows como Linux. Puesto que Java no se podía utilizar y es el lenguaje que permite una mejor portabilidad de interfaces gráficas, se decidió no crearla.

En una ampliación sería imprescindible hacer la interfaz gráfica, ya fuera ad hoc para un sistema operativo o procurar usar librerías que permitieran generar GUI que estuvieran disponibles para Windows y Linux como sería el caso de QT 4.

En cualquier caso, lo esencial sería disponer de una interfaz amigable y usable que permitiera gestionar todas las funcionalidades de gestión de la tarjeta de forma sencilla, como, por ejemplo, mediante el uso del ratón.

La interfaz tendría la finalidad de ahorrar tiempo a la hora de ejecutar la aplicación y aumentar su usabilidad. Un diseño sencillo, pero eficiente, consistiría en un cuadro central, donde se mostrarían todos los objetos accesibles de la tarjeta, con una serie de botones en la parte inferior con las principales acciones a realizar, un menú en la parte superior con las opciones más generales a nivel de la aplicación y menús emergentes (botón derecho del ratón) que ofrecieran opciones para interactuar con los objetos (como podrían ser creación, eliminación, uso, etc.).

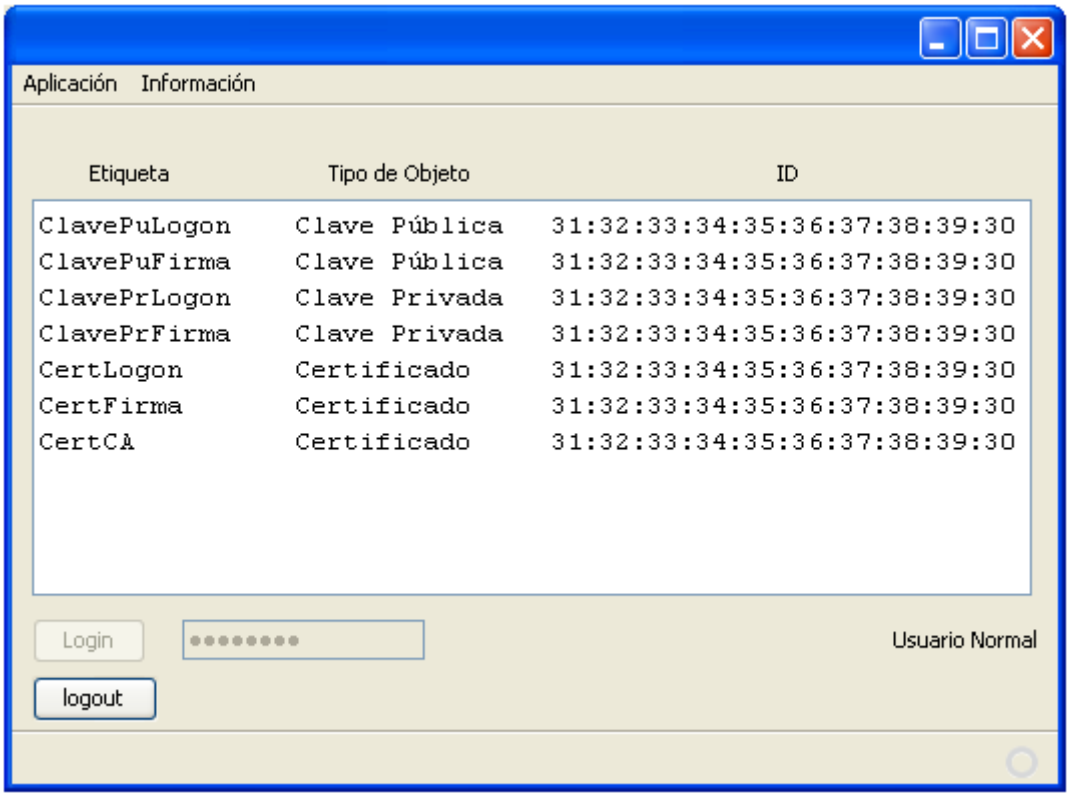

Ilustración 53: GUI aplicación

La imagen del panel de la aplicación fue generada con NetBeans y realmente es un GUI java. La versión final debería ser más elaborada, pero en este proyecto sólo se pretende plasmar la necesidad de un GUI a futuro y se ha hecho un esbozo de cómo podría estar distribuida en pantalla.

7.2 – IMPLEMENTACIÓN ORIENTADA A OBJETOS

Pese a que la implementación realizada (programación estructurada) es adecuada para el problema planteado en este proyecto, una implementación orientada a objetos siempre es algo bastante útil.

Aunque el dominio del problema (principalmente PKCS#11) es estático y no va a sufrir cambios a largo plazo, tener una implementación orientada a objetos permite un mantenimiento de la aplicación más sencillo. Que el dominio no sea cambiante no significa que las necesidades de la aplicación no lo sean y, por tanto, una implementación orientada a objetos facilita enormemente todas las modificaciones o ampliaciones de la aplicación.

Una orientación a objetos permitiría, además, añadir nueva semántica al sistema (relaciones entre clases) y, además, permitiría hacer una gestión más limpia de los errores a través de las excepciones.

Por último, decir, que una implementación orientada a objetos facilitaría bastante la creación de una interfaz gráfica de usuario y su acoplamiento con el núcleo del sistema.

A modo de tentativa (no se trata de una versión final), se presenta un esbozo de cómo podría estar diseñada la aplicación con una orientación a objetos. Para ello, se presenta, a continuación, el diagrama de clases que podría tener el sistema implementado bajo este paradigma.

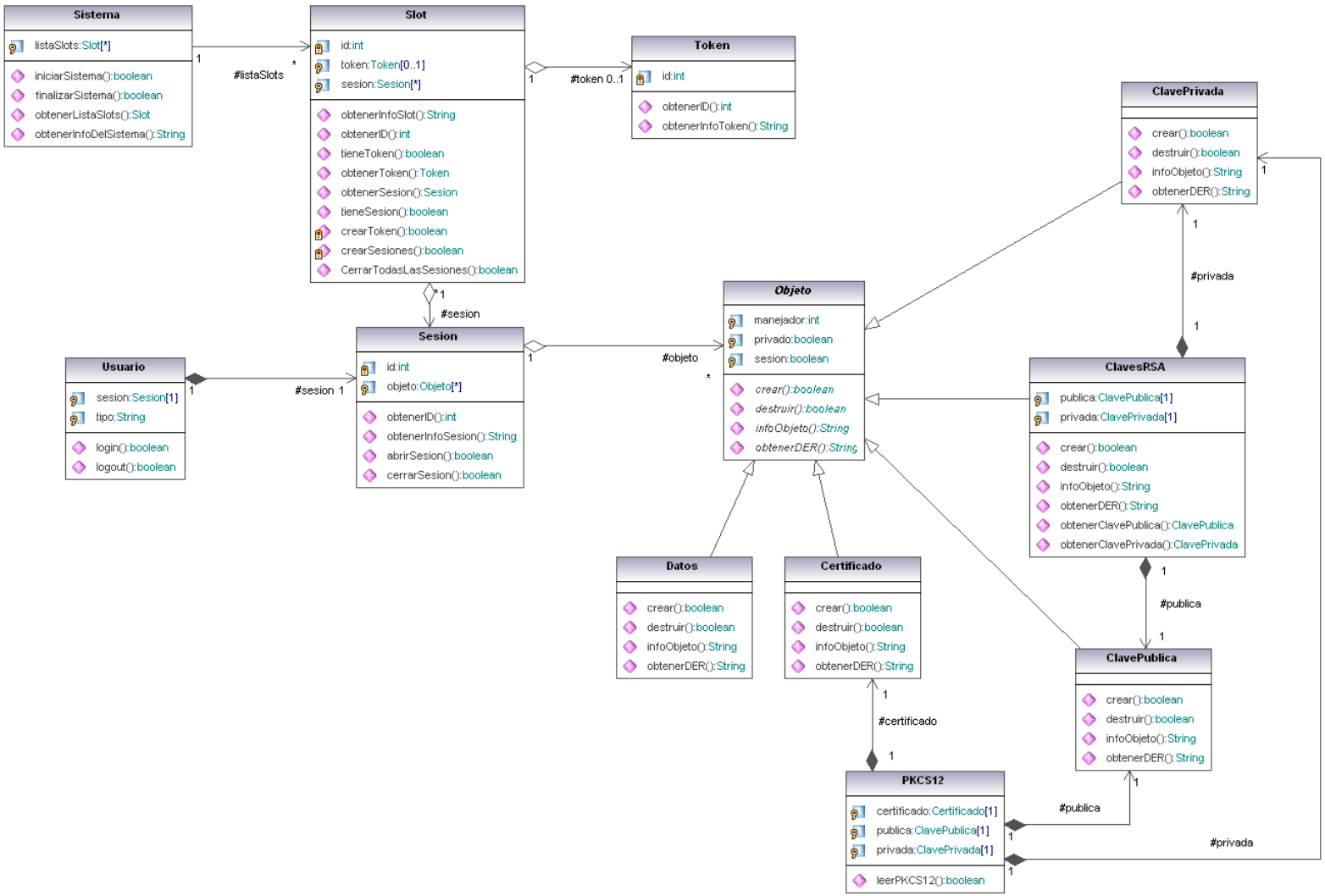

Ilustración 54: Esbozo de un diagrama de clases

7.3 – LENGUAJES DE MARCADO

En este proyecto, hay una característica que, para su correcto funcionamiento, depende del conocimiento de una información extra. Se trata de las firmas digitales. Para hacer las firmas genéricas y que se puedan adaptar a cualquier dominio y extensión (sobre todo a cambios a posteriori), no se realiza ningún tratamiento sobre ellas. En otras palabras, las firmas son escritas en fichero con una longitud de 128 bytes (si la clave privada era de 1024 bits) o de 256 bytes (en caso de claves de 2048 bits). Estos ficheros no contienen ninguna información extra, solamente los bytes que componen la firma.

El problema que se plantea con las firmas viene por dos motivos: el primero es que para comprobar la firma se necesita conocer el mecanismo que se utilizó para firmar, ya que, aun conociendo la clave privada correcta, es necesario conocer el mecanismo de firma para poder verificarla. El segundo motivo es conocer en sí la clave privada que fue utilizada para firmar. Este segundo problema no tiene solución factible, pero se pueden hacer aportes para que, aunque sea de forma local, se pueda facilitar el proceso de identificación de la clave.

Para facilitar todo esto, una posible solución sería no escribir las firmas directamente en un fichero, sino formatearlo todo dentro de una estructura como podría ser el caso de XML. Se podría plantear escribir la firma como un campo más de un árbol XML y, además, añadir más nodos con información adicional que permitiera conocer datos importantes, como por ejemplo, el mecanismo utilizado para firmar.

Un ejemplo de DTD que podría representar el documento de la firma podría ser:

```
<!ELEMENT Firma (Documento, Firmante, MecanismoFirma,
IDClaveFirma, ValorFirma)>
<!ELEMENT Documento (#PCDATA)>
<!ELEMENT Firmante (#PCDATA)>
<!ELEMENT MecanismoFirma (#PCDATA)>
<!ELEMENT IDClaveFirma (#PCDATA)>
<!ELEMENT ValorFirma (#PCDATA)>
```

`Documento` representa el fichero que fue firmado. `Firmante` es un campo que contiene algún tipo de referencia sobre la persona que firmó. `MecanismoFirma` sería el mecanismo utilizado para firmar y ayudaría a evitar confusiones. `IDClaveFirma` sería el ID interno del objeto que firmó (la clave privada). Este ID sólo sirve para que el usuario verifique sus propias firmas, ya que el ID de los objetos cambia de una tarjeta a otra aunque el objeto sea el mismo. `ValorFirma` contendría los bytes de la firma digital del documento.

Un ejemplo de ficheros XML que segaría el anterior DTD sería:

```
<Firma>
      <Documento>PFC.docx</Documento>
      <Firmante>Ignacio Álvarez Santiago</Firmante>
      <MecanismoFirma>MD5_RSA_PKCS</MecanismoFirma>
      <IDClaveFirma>
            31:32:36:35:39:37:36:33:31:34
      </IDClaveFirma>
      <ValorFirma>
            badafe924d05013a8bbabf734b5bb85b299e3483faf4e34d36aec
            c8fef573f07c0d9748b048be507d1aa682bfaf883a66ead1dc83a
            bdbb0a3319f6995de0410e99b7989eaefb5a0964f873a6f9aa6a0
            c8632d20e466e75b889e288162f38fbdba3f48426181d0552114f
            011e81c1afe622a60476f1fa0f13aa58fa3a918bdc84306d4d252
            453226cd328e5ae2f9f185b32a816648ce8333256c5e911a0a9ea
            fec91a7de3a45dd2bcf151001604e43b1f79ae63d02fdb5869487
            687058d926bbf07392ba3748e0f52e803672e7933c7f234d25d9b
            94865bcf1502a76217a7885a582ee341425f07911bab3aeaaf154
            91e7cd7563908063e711891af94ad8ea2a4
      </ValorFirma>
</Firma>
```

Como se puede observar a simple vista, esta estructura sólo sirve como guía para facilitar el proceso de verificación de firmas. No aporta ningún tipo de seguridad y nunca debería descartarse una firma al no poderse verificar utilizando este sistema XML (no hay garantías, por ejemplo, de que no se hay cambiado el mecanismo en el documento XML). Además, la identidad no se garantiza de ningún modo.

Este mecanismo sólo pretende facilitar el proceso de verificación de las firmas dentro de un marco seguro, pero no provee de dicho marco. Para garantizar la seguridad de éste mecanismo, se deberían usar:

- Sumas de verificación. Garantizarían que el mensaje XML es íntegro.
- Codificación mediante clave simétrica del mensaje XML. Garantizaría que sólo aquellos que comparten el secreto podrían ver, utilizar o modificar el mensaje XML.
- Firma del XML. Mediante la firma con una clave privada concreta, que todo receptor de la misma supiera de ante mano qué clave pública utilizar para verificarla (en caso de haber más de una). Además, tendría que convenirse, previamente, qué mecanismo utilizaría dicha firma, por ejemplo, RSA con SHA1.

Sin la utilización de alguna de esas ideas (o de varias en conjunción) el XML es una herramienta que ayuda a verificar firmas en entornos seguros, pero no en otros entornos.

7.4 – SOPORTE PARA CA

Debido a que el proyecto ya tiene soporte para certificados X.509 y además gestiona CSR un paso sencillo de dar y que apretaría otra vía de ampliación a futuro para el proyecto, sería la inclusión de soporte para CAs a través de OpenSSL.

El proyecto permitiría crear un certificado autofirmado raíz con los datos con las restricciones necesarias para ser una CA, además de su par de claves. La tarjeta funcionaría de almacén de la CA, ay que su clave privada no podría ser extraída.

Se suministraría toda la funcionalidad necesaria para gestionar la firma de certificados digitales, listas de revocación etc. La aplicación haría las veces de servidor y base de datos de toda la información relacionada con la CA, mientras que la tarjeta sería el elemento portable capaz de realizar las operaciones, ya que contendría la clave privada de la CA.

Toda la funcionalidad para gestionar una CA como la que ofrece la aplicación openssl.exe de Windows (o su equivalente de Linux) está ofrecida en la API de OpenSSL, de modo que una vez conocidos esos mecanismos gestionar una CA sería relativamente sencillo.

8 – GLOSARIO

.NET: Plataforma de desarrollo de Microsoft, enfocada al desarrollo de programas para sistemas Windows. Se caracteriza principalmente por los lenguajes Visual C++, Visual Basic, C# y ASP.

AES: Es el estándar de criptografía simétrica y el más seguro de todos los algoritmos de cifrado simétrico que se usan comúnmente. Tiene versiones que van desde los 182 bits a los 256 (mucho más que triple DES).

API: Application Programming Interface (Interfaz para la Programación de Aplicaciones). Es una interfaz donde se expresan funcionalidades y cómo pueden ser invocadas por una aplicación de terceros para obtener la funcionalidad que ofrece.

Array: Colección de elementos de un mismo tipo almacenados en memoria de forma consecutiva y a los que se puede acceder por medio de un índice. Generalmente se asocia al concepto matemático de vector 8array unidimensional).

BER: Basic Encoding Rules (Reglas de Codificación Básicas). Son unas reglas de codificación basadas en secuencias que contienen los tipos primitivos, expresando toda secuencia o tipo su longitud y el tipo de información que contiene.

BNF: Backus Normal Form o Forma Normal de Backus. Es un formalismo de expresión de gramáticas recursivas que permite definir como se generan palabras pertenecientes al universo de dicha gramática.

Buffer: Fragmento de memoria RAM consecutiva de tamaño prefijado que es utilizada como memoria intermedia de lectura o escritura.

C: Es un lenguaje de programación estructurado que permite el uso de características bajo nivel, enfocado a la programación de sistemas, donde el programador tiene gestión total de la memoria. ANSI hace referencia a la parte estandarizada del lenguaje.

C++: Es la versión Orientada a Objetos de C. Su sintaxis es similar y la novedad es que permite crear diseños Orientados a Objetos.

CA: Certificate Authority (Autoridad de Certificación) es el elemento básico de confianza en una arquitectura PKI. Su funcionalidad principal es la de firmar certificados y mantener las listas de revocación.

Certificado Digital: Es un documento digital que sirve para exportar la clave pública de un usuario y que pueda ser utilizada de forma segura por terceros.

Consola: Es el terminal del sistema. Es una interfaz que permite ejecutar programas en modo texto (no permite gráficos). En Windows el intérprete de la consola se ejecuta por medio de la aplicación cmd.exe.

CSP: Son las siglas de Criptographic Service Provider (proveedor de servicios criptográficos). Es una aplicación (generalmente un DLL) diseñada para que el sistema operativo Windows pueda interactuar con las tarjetas a través de una interfaz conocida. Suele utilizarse para cargar información relativa a logon con tarjetas, generación de claves, firma, etc.

CSR: Son las siglas de Certificate Signing Request (Petición de Firmado de Certificado). Es el primer paso para la construcción de certificados X509. Una CSR es el certificado tal cual lo genera el usuario con su clave pública, pero requiere de la firma de una autoridad (una CA) para convertirse en un certificado exportable de confianza.

DER: Distinguished Encoding Rules (Reglas de Codificación Distinguidas. Son un subconjunto de las reglas BER, de modo que toda codificación que siga las reglas DER forzosamente es BER.

DES: Es un algoritmo simétrico de cifrado creado por IBM que sustituyó a DES. Básicamente es un DES mejorado, que da varias pasadas en el proceso de cifrado y que usa una clave de mayor tamaño. (DES usa 56 bits efectivos y triple DES 112bits)

DLL: Son librerías de código dinámico (Dinamic Link Library o Librería de Enlace Dinámico), lo que significa que su código (previamente compilado) es ejecutado bajo la demanda del programa llamador.

DNIe: Son las siglas de Documento Nacional de Identidad electrónico. Es el nuevo modelo de DNI implantado en España que incorpora un chip criptográfico (el mismo de las tarjetas Ceres). Incorpora certificados y claves que permiten la autenticación del usuario frente a la administración en Internet.

DTD: Son las siglas de Document Type Definition o Dedición de Tipo de Documento. Es un lenguaje que permite definir estructuras de documentos XML.

Firefox: Es un popular navegador web libre (con versiones para todos los sistemas operativos) muy potente, caracterizado por su elevada velocidad de carga y por su ampliación de funcionalidad mediante el uso de plugins.

GANTT: Es un tipo de diagrama en el que se pueden ver fácilmente las diferentes tareas a realizar dentro de un proyecto, la duración, las dependencias entre las tareas y los recursos necesarios para realizar cada una de ellas.

GUI: Son las siglas de Grafical User Interface en castellano Interfaz Gráfica de usuario. Una GUI es un programa que actúa de interfaz entre el núcleo de la aplicación y el usuario, permitiéndole a éste interactuar con la aplicación de forma gráfica.

HTTPS: HyperText Transfer Protocol Secure (Protocolo de Transporte de HiperTexto Seguro). Es la versión cifrada y segura del protocolo básico web HTTP.

Interfaz de Usuario: Es la capa de presentación visual (gráfica o textual) que permite al usuario, de forma más o menso intuitiva, ejecutar la funcionalidad que ofrece la aplicación.

Java: Lenguaje de programación orientado a objetos independiente de la plataforma. Permite compilar un programa y que pueda ser usado en cualquier arquitectura y sistema operativo. Requiere de una máquina virtual ya que la compilación no genera código máquina sino un código intermedio (bytecode).

Login: Tiene un significado similar a logon. Hace referencia a entrar dentro del sistema, en este caso, la tarjeta.

Logon: El término logon hace referencia a acceder al sistema. En Windows hacer logon supone poder comenzar a utilizar el sistema operativo con los permisos prefijados correspondientes al usuario con el que se hace logon. Normalmente el logon se hace suministrando un nombre de usuario y su contraseña. Con SmartCards el logon se realiza suministrando la tarjeta al sistema (a través de un lector) y escribiendo el PIN de la tarjeta.

MD5: Es una función resumen muy extendida (soportada por todos los sistemas PKI) que obtiene un resumen de un documento a base de realizar operaciones lógicas. El tamaño de los resúmenes es de 128 bits.

OpenSC: Es un proyecto libre enfocado a la gestión de las SmartCards tanto en su aspecto funcional (PKCS#11) como en su aspecto lógico interno (PKCS#15).

PEM: Privacy-Enhanced Mail o Privacidad en el Correo Mejorada. Es una codificación Base 64 ideada para el envío de correos con codificaciones no compatibles con ASCII 7 bits.

PFX: Son la representación de más alto nivel dentro de PKCS#12. Son el conjunto ASN.1 con toda la información contendía dentro del PKCS#12 y los metadatos asociados para que sea entendible. PFX es también la extensión habitual de los ficheros que contienen perfiles PKCS#12.

PKCS#11: Es el estándar de RSA (PKCS significa Public Key Criptographic Standard o Estándar Criptográfico de Clave Pública) de criptografía pública encargado

de definir como se debe interactuar con tokens criptográficos (comúnmente SmartCards).

PKCS#12: Es el PKCS definido por RSA de cómo debe ser exportada la información personal (ya que los PKCS#12 contienen certificados y claves del usuario). Al contener claves privadas y por motivos de seguridad suelen ser ficheros encriptados con criptografía simétrica.

PKCS#15: Este es el estándar de RSA que define cómo debe estructurarse la información dentro de un token criptográfico. En otras palabras, defines cómo debe ser la estructura de ficheros y la organización de la información dentro de la tarjeta.

PIN: Es la contraseña necesaria para poder acceder como usuario a la tarjeta.

PKI: Son las siglas de Public Key Infraestructure (Infraestructura de Clave Pública) y es básicamente la forma de denominar a toda la infraestructura de autoridades de certificaciones, certificados de usuario y "relaciones de confianza" que se establecen entre los distintos elementos, basadas en el criptosistema de criptografía asimétrica (clave pública y privada).

Plugin: Es una pequeña aplicación que se integra en otra más grande para aportarle nueva funcionalidad.

RSA: En criptografía pública, se refiere a un algoritmo concreto que es utilizado tanto para el cifrado asimétrico como para la firma electrónica. Es, en la actualidad, el sistema más utilizado en la criptografía de clave pública en el mundo.

SHA1: Es una función de resumen cuyo funcionamiento es similar a MD5, aunque el algoritmo de resumen sea diferente. Es más lenta de realizar que MD5, pero, como ventaja, tiene que los resúmenes son de 160 bits.

Slot: Traducido como ranura o rendija, hace referencia a la "dirección" dentro del sistema del lector de tarjetas. Se deben diferenciar de algún modo, ya que una misma máquina puede tener varios lectores de tarjetas a la vez.

SmartCards: Son tarjetas con chip integrado (en español también se las denomina tarjetas inteligentes). Una tarjeta ha de tener una de las siguientes capacidades (o ambas) para poder ser considerada una SmartCard, ha de tener capacidad de almacenamiento o capacidad de procesamiento (generalmente criptográfico).

TAD: Tipo Abstracto de Datos. Es un conjunto de operaciones que actúan sobre unos datos concretos con el fin de modificarlos u obtener información de los mismos. El concepto de TAD está estrechamente ligado con las estructuras de datos. Algunos TAD famosos son las listas enlazadas, las pilas y las colas.

Token: Traducido como ficha, es un sinónimo en este proyecto de SmartCard.

UML: Unified Modeling Languaje o lenguaje de modelado unificado es un estándar de facto para la representación del software. Con él, se pueden realizar diagramas fácilmente entendibles (de clases, de secuencia, de casos de uso, de actividad, etc.) por todo el mundo que domine el lenguaje y expresar todo lo necesario relativo al análisis y diseño del software.

UNIX: Familia de sistemas operativos representados en la actualidad principalmente por Linux. Los sistemas Unix son generalmente abiertos y están portados a la inmensa mayoría de las arquitecturas.

X.509: Es un estándar de certificado utilizado para exportar las claves públicas, donde se define quien es el propietario de las mismas, la cadena de confianza con las autoridades y una serie de informaciones como el uso al que están destinadas las claves públicas.

XML: Acrónimo de eXtensible Markup Language o Lenguaje de Marcado Extensible. Es un lenguaje etiquetado que permite crear documentos estructurados.

9 – BIBLIOGRAFÍA

- **Página Web de RSA sobre PKCS#10:** http://www.rsa.com/rsalabs/node.asp?id=2132
- **Página Web de RSA sobre PKCS#11:** http://www.rsa.com/rsalabs/node.asp?id=2133
- **Página Web de RSA sobre PKCS#12:** http://www.rsa.com/rsalabs/node.asp?id=2138
- **Página Web de RSA sobre PKCS#15:** http://www.rsa.com/rsalabs/node.asp?id=2141
- **Página Web del proyecto OpenSSL** http://www.openssl.org/
- **Página Web para descargar OpenSSL para Windows:** http://www.slproweb.com/products/Win32OpenSSL.html
- **Página Web de una Cross Reference de OpenSSL:** http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/common/openssl/
- **Página Web del Cross Reference de Firefox:** http://mxr.mozilla.org/firefox/
- **Página Web del MSDN del laboratorio de informática de la UC3M:** http://www.lab.inf.uc3m.es/servicios/msdn
- **Página Web de Microsoft para la realización de logon con SmartCard:** http://support.microsoft.com/kb/281245
- **Página Web de Microsoft sobre PKI en Windows Server 2003:** http://www.microsoft.com/windowsserver2003/technologies/pki/default.mspx
- **Página Web de MinGW:** http://www.mingw.org/
- **Página Web de NetBeans** http://netbeans.org/
- **Página Web de NetBeans para C:** http://netbeans.org/features/cpp/
- **Página de Wikipedia.** http://es.wikipedia.org/wiki/Wikipedia:Portada

- Martin Fowler. **UML Distilled Third Edition.** Addison-Wesley. 2004.

- Matt Messier, John Viega. **Secure Programming Cookbook.** O'Reilly. 2003.

- Pravir Chandra, Matt Messier, John Viega. **Network Security with OpenSSL.** O'Reilly. 2002.

- RSA Laboratories**. PKCS#10 v1.7: Certification Request Syntax Standard**. 2000.

- RSA Laboratories. **PKCS#11 Base Functionaly v2.30: Cryptoki – Draft 4**. 2009.

- RSA Laboratories. **PKCS#11 Mechanisms v2.30: Cryptoki – Draft 7.** 2009.

- RSA Laboratories**. PKCS#12 v1.0: Personal Information Exchange Syntax.** 1999.

- RSA Laboratories. **PKCS#15 v1.1: Cryptographic Token Information Syntax Standard.** 2000.

APÉNDICES

ESTUDIO DEL DESARROLLO DEL PROYECTO

En este punto se describen las fases por las que ha pasado la realización de este proyecto (explicando qué se ha hecho en cada una de ellas). Para hacer más visible toda la información suministrada, se presentará un diagrama Gantt donde se visualizarán de forma gráfica las fases, las tareas que componen las fases, las dependencias entre las mismas (dependencias de orden) y el tiempo empleado desde que se inicio cada fase o tarea hasta que se concluyó.

Las fases descritas en las que se dividió en proyecto son las siguientes:

	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1	**ANÁLISIS**	**25 días**	**mar 01/09/09**	**lun 05/10/09**	
2	Requisitos Funcionales	20 días	mar 01/09/09	lun 28/09/09	
3	Requisitos no Funciona	15 días	lun 07/09/09	vie 25/09/09	
4	Elección plataforma	5 días	mar 29/09/09	lun 05/10/09	2
5	Elección Interfaz	5 días	mar 29/09/09	lun 05/10/09	2
6	**APRENDIZAJE**	**25 días**	**mar 06/10/09**	**lun 09/11/09**	**1**
7	Aprendizaje PKCS#11	25 días	mar 06/10/09	lun 09/11/09	
8	Aprendizaje OpenSSL	18 días	lun 12/10/09	mié 04/11/09	
9	Aprendizaje de Soporte	8 días	lun 12/10/09	mié 21/10/09	
10	**DISEÑO**	**27 días**	**mar 10/11/09**	**mié 16/12/09**	**6**
11	Elección Método de Dis	4 días	mar 10/11/09	vie 13/11/09	
12	Diseño Componentes P	23 días	lun 16/11/09	mié 16/12/09	11
13	Diseño Componentes C	19 días	lun 16/11/09	jue 10/12/09	11
14	Diseño Componentes d	8 días	lun 16/11/09	mié 25/11/09	11
15	**IMPLEMENTACIÓN**	**45 días**	**jue 17/12/09**	**mié 17/02/10**	**10**
16	PKCS#11	45 días	jue 17/12/09	mié 17/02/10	
17	OpenSSL	30 días	lun 21/12/09	vie 29/01/10	
18	Soporte	15 días	jue 17/12/09	mié 06/01/10	
19	Interfaz	5 días	jue 31/12/09	mié 06/01/10	
20	**PRUEBAS**	**35 días**	**jue 31/12/09**	**mié 17/02/10**	**10**
21	Unitarias	31 días	jue 31/12/09	jue 11/02/10	
22	Rendimiento	25 días	vie 08/01/10	jue 11/02/10	
23	Aceptación	20 días	jue 21/01/10	mié 17/02/10	
24	DOCUMENTACIÓN	60 días	jue 17/12/09	mié 10/03/10	10

Ilustración 55: Fases del proyecto

Descripción de las tareas

Análisis

En la fase de análisis se recogieron todas las necesidades del proyecto, todo lo que el proyecto tenía que realizar. Esto se realizo en la tarea de Requisitos Funcionales. Además de los funcionales, también hubo que definir cómo debían hacerse ciertas tareas, restricciones a las mismas para ser exactos. Esto se realizó en la tarea de requisitos no funcionales.

Una vez se tenía claro lo que había que hacer en el proyecto y cómo debía ser hecho se eligió la plataforma para el desarrollo. Con plataforma se quiere englobar la elección del lenguaje y del sistema operativo donde se iba a implementar la aplicación. Esto más que una elección fue prácticamente un requisito impuesto. Una vez estaba definida la plataforma se eligió el formato de interfaz de usuario que se iba a utilizar (la de consola textual como ay se explico en el punto 4.4.1).

Todos estos aspectos del análisis fueron tratados junto al tutor para adaptarse de la mejor forma posible a las necesidades que se requerían para el proyecto.

Aprendizaje

En esta fase, que comenzó una vez concluido el Análisis del proyecto, fue donde se adquirieron los conocimientos necesarios para poder desarrollar el proyecto. Debido a que todas las tecnologías que se iban a utilizar para el desarrollo del proyecto eran nuevas (nunca vistas con anterioridad) fue una tarea ligeramente complacida. En primer lugar hubo que adquirir todos los conocimientos sobre el estándar PKCS#11 para poder utilizar su API para gestionar las SmartCards. Después hubo que aprender a manejar la API de OpenSSL que se trata de una API enorme y en muchos aspectos muy mal documentada. Esta tarea fue quizás la más tediosa, debido precisamente a la falta de información para realizar distintas operaciones.

Además de esas dos tareas de aprendizaje básicas descritas anteriormente, fue necesario adquirir otros conocimientos relacionados con PKI de Windows, sobre codificaciones y formalismos que eran necesarios para tener una mejor visión del desarrollo del proyecto. Algunas de estos conocimientos de soporte fueron vitales para poder acometer la implementación del proyecto, como fue el caso de la codificación DER.

Diseño

En esta fase de describió cómo debían ser generados todos los componentes y funcionalidades del sistema. Esta fase dio comienzo nada más concluir el periodo de aprendizaje de las tecnologías necesarias para el desarrollo del proyecto.

En primer lugar, se definió cómo se iba a diseñar el proyecto en función de todos lo tratado en el Análisis. Puesto que se había elegido C, Windows y una interfaz en consola, se adapto un diseño estructurado y basado en módulo para, a partir de esa elección, realizar el diseño concreto de cada uno de los elementos (componentes).

A partir de ahí se diseñaron tres grandes bloques. Por un lado se diseñan todos los componentes que tenían relación con PKCS#11. Una vez concluido, se diseñó el componente que iba a gestionar OpenSSL. A la par de los dos se fueron diseñando diversos componentes (como la lista) encargados de ofrecer soporte a los dos anteriores.

Implementación

Esta fue, seguramente, la fase más complacida de todo el proyecto. En ella se programó toda la aplicación en base a lo descrito en el Diseño realizado. En especial fue difícil conseguir ejecutar de forma satisfactoria las operaciones sorbe la tarjeta (todo lo que tenía que ver con PKCS#11). Para ello hubo que consultar en alguna ocasión cómo acometían estas tareas otras aplicaciones genéricas a las que se tenía acceso al código fuente, como fue el código fuente de Mozilla Firefox 3. Esto se realizó sobre todo para conseguir obtener plantillas genéricas para realizar ciertas tareas (como las de generar objetos). Por desgracia, el código de Firefox no es demasiado genérico y no era 100% compatible con las tarjetas utilizadas, así que no quedó más remedio que probar diferentes alternativas hasta que se dio con la más genérica que se adaptaba a todas las tarjetas utilizadas para este proyecto.

La complejidad de OpenSSL no fue tanto la implementación sino saber cómo implementarlo, ya que su API es muy grande (algo bueno ya que permite realizar gran cantidad de operaciones) pero, por desgracia, está bastante mal documentada por lo que conseguir implementar funcionalidad con OpenSSL era muy lento. Debido a que algunas operaciones que se querían realizar y que no se encontró en la API cómo realizarlas, fue necesario mirar el código fuente de OpenSSL para estudiar como hacía ciertas operaciones a bajo nivel para así poder seguir una técnica similar en el proyecto. Esto fue necesario, por ejemplo, para poder firmar los CSR generados en la

tarjeta con la clave privada que se complementaba con la pública incluida en el CSR en sí.

La implementación de los otros componentes, así como la de la interfaz, no supusieron complicación alguna, ya que se trataba de operaciones simples realizadas muchas veces antes. En el caso de la interfaz se trató solamente de validar parámetros (lectura de enteros desde cadenas, validar opciones, etc.) y formatear cadenas para mostrarlas de forma ordenada por pantalla.

Pruebas

Aunque no se ha creado un punto en esta memoria que describa las pruebas, se realizaron tres tipos de pruebas que garantizaran el correcto funcionamiento de todo lo implementado.

En primer lugar están las pruebas "unitarias", estas pruebas consintieron simplemente en comprobar que todas las funciones implementadas, ofrecían resultados correctos para todo tipo de parámetros de entrada.

El siguiente tipo de pruebas, fueron las de rendimiento. Estas pruebas se centraron en comprobar básicamente el consumo de memoria, para garantizar que no había reservas de memoria sin liberación que hicieran que el tamaño en RAM del programa creciera sin control. Esto fue importante ya que en C esta gestión la realiza el programador y un fallo puede ser fatal.

El último tipo, fueron las pruebas de aceptación. Se entiende por esto, la realización de pruebas de uso de la aplicación que determinen si la funcionalidad implementada se comporta de forma correcta, como podría ser, por ejemplo, comprobar que la aplicación realiza de forma adecuada la firma digital.

Documentación

La documentación es lo que se presenta en esta memoria. En ella se recogen todos los puntos realizados tales como Análisis, Diseño (con matices de la implementación), definición de las tecnologías sudas, así como todo lo que se ha tratado en la presente memoria.

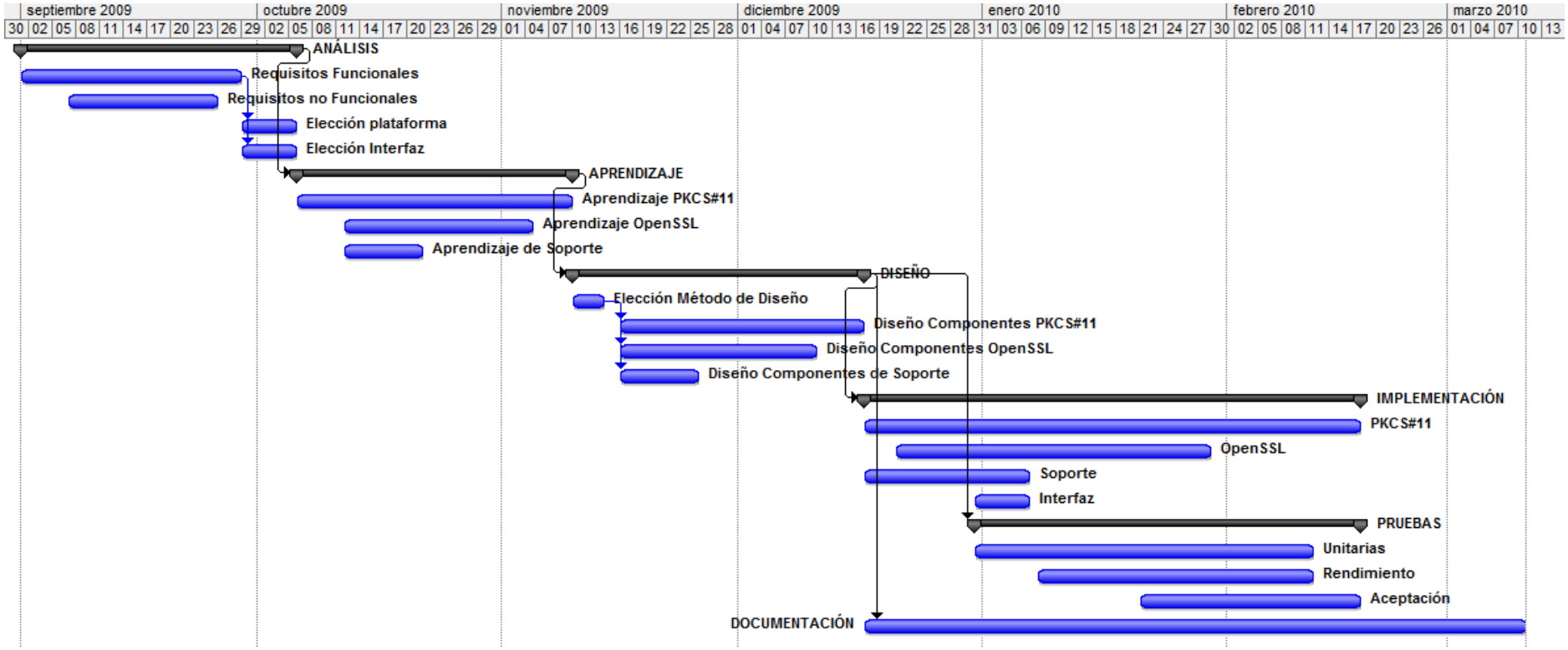

Ilustración 56: Gantt del proyecto

SOFTWARE UTILIZADO

SISTEMAS OPERATIVOS

Windows XP

Es el sistema operativo base utilizado para desarrollar este proyecto. En general, todas las aplicaciones necesarias para el desarrollo o para la documentación y los diferentes módulos PKCS#11 de las tarjetas funcionan correctamente bajo este sistema.

Es un sistema operativo que, en comparación con las versiones de Windows más modernas, consume pocos recursos y, por tanto, puede ser instalado en máquinas más antiguas con menos capacidad hardware.

Puesto que era imperativo que el desarrollo en un principio fuera realizado bajo Windows, se ha utilizado esta versión XP por los motivos antes mencionados.

La versión utilizada es Windows XP SP3 32 bits obtenida del MSDN de la Universidad Carlos III de Madrid.

Windows Server 2003

Otra necesidad del proyecto era generar CSR que pudieran ser firmados por una CA para que los certificados X.509 obtenidos fueran válidos para realizar logon, concretamente el requisito era realizar logon en Windows Server 2003. Por tanto, fue necesario instalar un Windows Server 2003 para poder realizar todas las pruebas y comprobaciones y para verificar que, tanto los CSR generados, como los certificados obtenidos, eran válidos para hacer logon sobre el sistema con una SmartCard.

Debido a que se trataba de un sistema para pruebas, que no iba a usarse para nada más y que se trata de una versión servidora de Windows, no se realizó una instalación física en una máquina. En su lugar, se instaló en una máquina virtual VMWare.

La versión utilizada fue Windows Server 2003 Enterprise Edition R2 de 32 bits obtenida del MSDN de la Universidad Carlos III de Madrid.

OTRO SOFTWARE

Cryptoki

Dentro del Cryptoki se engloban todos los módulos PKCS#11 y aplicaciones de soporte suministradas por cada una de las tarjetas utilizadas para poder interactuar con ellas.

Todas las tarjetas incluyen dos formatos de interacción: el primero es la DLL que contiene el módulo PKCS#11 de dicha tarjeta que permite a aplicaciones de terceros interactuar con ella. El otro sistema de interacción (que no hay sido utilizado en este proyecto) es el CSP. Todas las tarjetas proveen un CSP para permitir que el sistema operativo Windows interactúe con ellas (por ejemplo para realizar logon).

Cada tarjeta, además, incluye cierto software que permite realizar operaciones muy sencillas sobre ella, como podrían ser: desbloqueos o cambios de PIN, importación de certificados, visualización de elementos, etc. En algunos casos, estas aplicaciones no siguen el estándar PKCS#11 y el Cryptoki no provee ciertas funcionalidades, de modo que es necesario utilizar estas aplicaciones para realizar algunas de esas operaciones. Un ejemplo sería el desbloqueo de PIN en tarjetas CERES.

Finalmente, dentro del Cryptoki, se incluyen también los 4 ficheros de cabecera mencionados en el diseño que son suministrados por RSA (disponibles en la página PKCS#11 de RSA, ver referencias).

OpenSSL

OpenSSL ya fue descrito en el apartado 2.7. Mencionar solamente que para la implementación de la aplicación se utilizó la DLL (y sus ficheros de cabecera) que implementa la API. Además, se utilizó OpenSSL como aplicación para usarlo como gestor de una CA que permitiera firmar los CSR generados en la tarjeta y obtener certificados genéricos válidos (OpenSSL no se usó para los certificados de logon. Para esto, se usó la CA de Windows Server 2003).

MinGW

MinGW es la versión del compilador GNU de C para Windows. Es el acrónimo de Minimalist GNU for Windows. Su principal característica es la implementación para Windows del compilador GCC. Se ha utilizado este compilador para obtener la aplicación por varios motivos: el primero es que, al tratarse de una versión portada de Unix a Windows, garantiza que los códigos ANSI C compilados bajo Windows sean fácilmente portados posteriormente en Linux (siempre que no se haga uso de características o librerías propias de Windows). La otra es que se trata de un gran compilador que provee de todo tipo de funcionalidades como pueden: ser la compilación para diferentes arquitecturas, atención de código máquina usable por el debugger (concretamente el GDB), obtención de código optimizado (versiones release), etc.

Además de todo lo mencionado MinGW, tiene otra gran ventaja: permite integrarse dentro del IDE NetBeans con lo que se tiene una excelente plataforma para el desarrollo de C dentro de Windows.

NetBeans

NetBeans es un IDE originalmente pensado para el desarrollo de Java, pero que, a través de sus plugins, ofrece una plataforma de desarrollo de C/C++. El compilador que correrá por debajo de NetBeans, como ya se ha dicho, será MinGW (un GCC). NetBeans permite interactuar con el GCC para compilar los códigos fuente, con GDB para poder depurarlos (con todas las características genéricas de depuración) y MAKE para poder definir una secuencia de compilación válida.

NetBeans con su complemento para C ofrece todas las características de su versión Java como pueden ser:

- Ejecución paso a paso y breakpoints
- Pila de llamadas
- Visionado de variables (incluso sobre el propio código fuente)
- Compilación y depuración
- Resaltado de sintaxis
- Sangrado automático y autocompletado de código
- Acceso a una definición desde el lugar donde se usa
- Acceso rápido a miembros (como en el caso de las estructuras)
- Visualización de todos los elementos del fichero fuente actual
- Definición de los parámetros de ejecución y compilación

Las versiones utilizadas en este proyecto de NetBeans han sido la 6.7.1 y la 6.8.

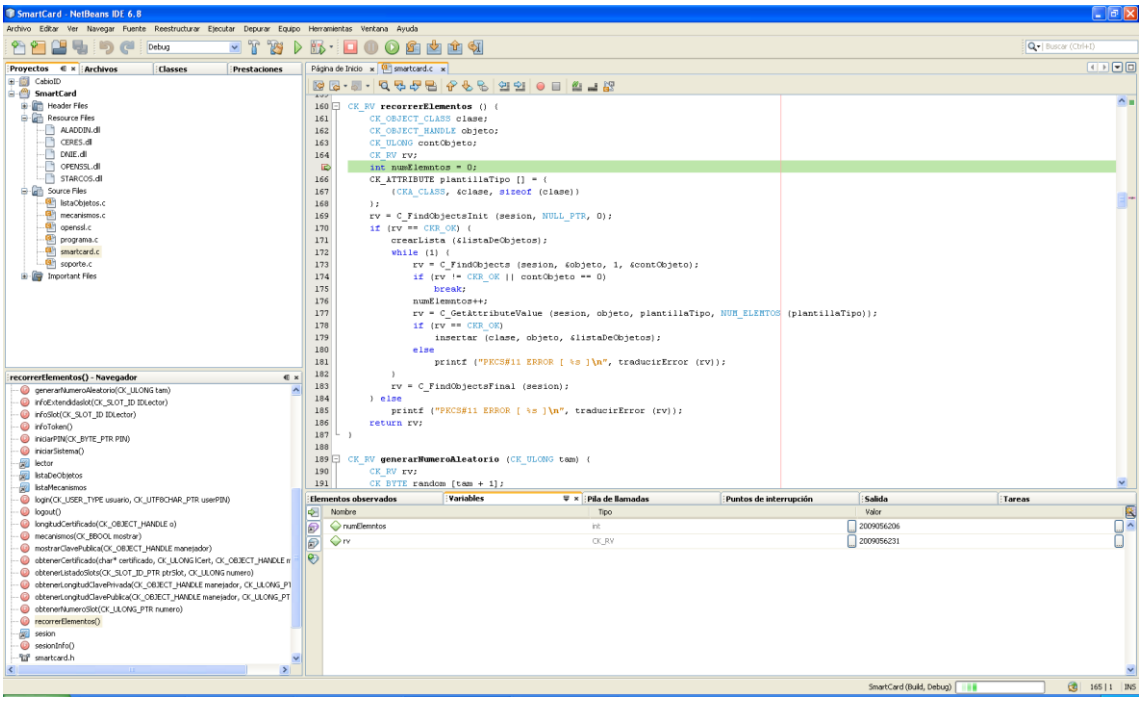

Ilustración 57: Pantalla de NetBeans

Microsoft Word

Es un procesador de textos de Microsoft principalmente para sistemas Windows que permite una buena edición de documentos de forma rápida y sencilla. Con este procesador, es con el que se está escribiendo la presente memoria. Sus principales características son:

- Formateo de texto sencillo (fuentes, párrafos, etc.)
- Creación de títulos e índices
- Creación de tablas y listas
- Edición de encabezados y pies de página
- Definición de estilos y plantillas
- Inclusión de imágenes
- Compatibilidad con objetos de otros programas Office (como Excel o Visio)
- Ortografía y gramática
- Previsualización del documento
- Guardar en distintos formatos (entre otros PDF).

En general, Word permite obtener resultados profesionales con poco esfuerzo y, además, se tiene siempre una imagen fiel del resultado final del documento.

La versión utilizada es Office 2007.

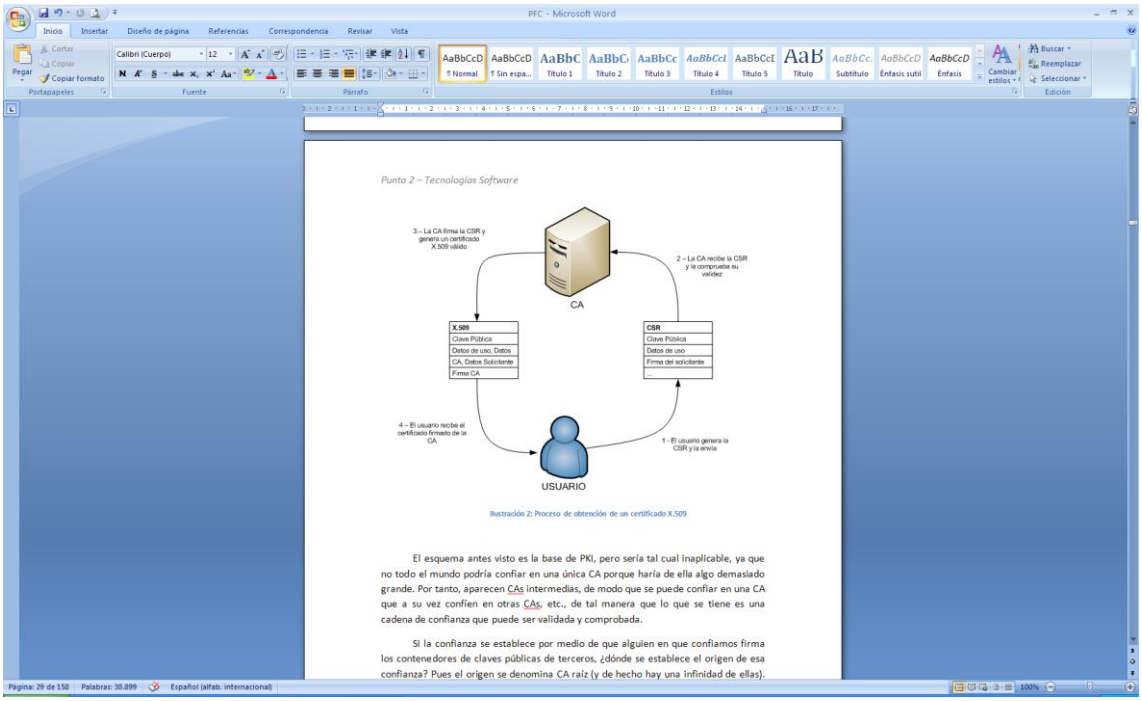

Ilustración 58: Pantalla de Microsoft Word

Microsoft Excel

Microsoft Excel es una potente hoja de cálculo que permite realiza gran cantidad de operaciones matemáticas y de contabilidad. En este proyecto, ha sido utilizado para la obtención de las gráficas de barras del rendimiento de las distintas tarjetas utilizadas. Al igual que en el caso de Word, la versión utilizada es la de Office 2007.

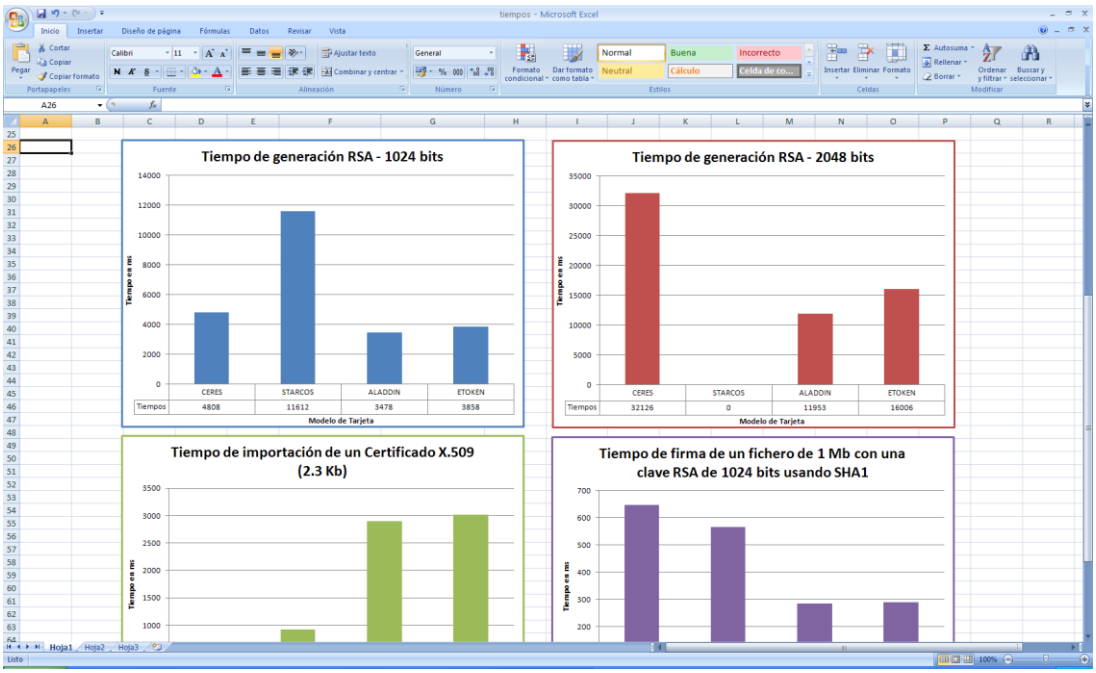

Ilustración 59: Pantalla de Microsoft Excel

Microsoft Visio

Visio es un editor de diagramas de Microsoft que, en general, es muy genérico (se pude hacer casi cualquier tipo de diagrama con él), pero, por desgracia, no permite dotar en muchos casos de gran calidad a los diagramas obtenidos (como el caso de concreto de diagramas UML). Visio se ha utilizado en este proyecto para la creación de los diagramas UML a excepción del de despliegue y el de componentes. La versión utilizada ha sido la de Visio 2007.

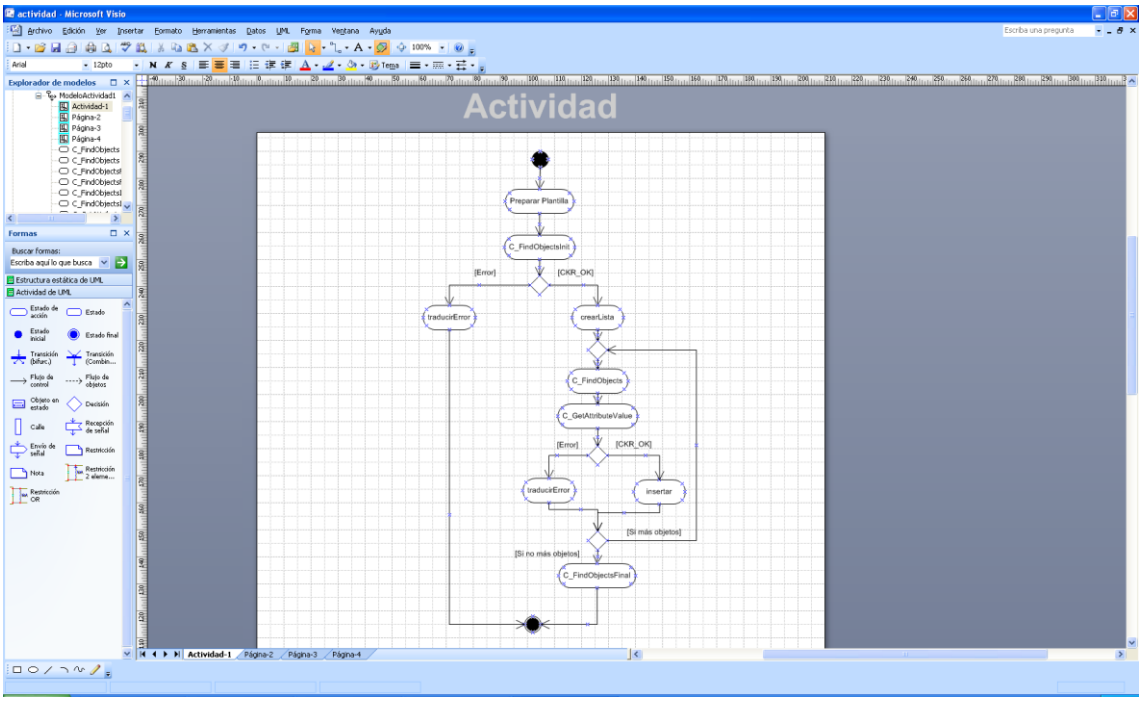

Ilustración 60: Pantalla de Microsoft Visio

Microsoft Project

Project ha sido la herramienta con la que se ha generado el diagrama Gantt y la especificación de las fases y tareas que componen el proyecto. Con esta herramienta se puede definir de forma rápida, duraciones de fases y tareas, establecer relaciones de subordinación o de precedencia, asignar recursos, etc.

En este proyecto sólo se ha definido un Gantt con las fases, sus relaciones y su duración. No se ha definido ni recursos empelados ni personal que se encargo de cada tarea (siempre fue la misma persona).

El resultado es un diagrama Gantt donde se puede hacer un fácil seguimiento de todas las tareas en las que se división el proyecto, que tareas se realizaron en paralelo y qué tareas fue necesario terminar antes de comenzar otras nuevas.

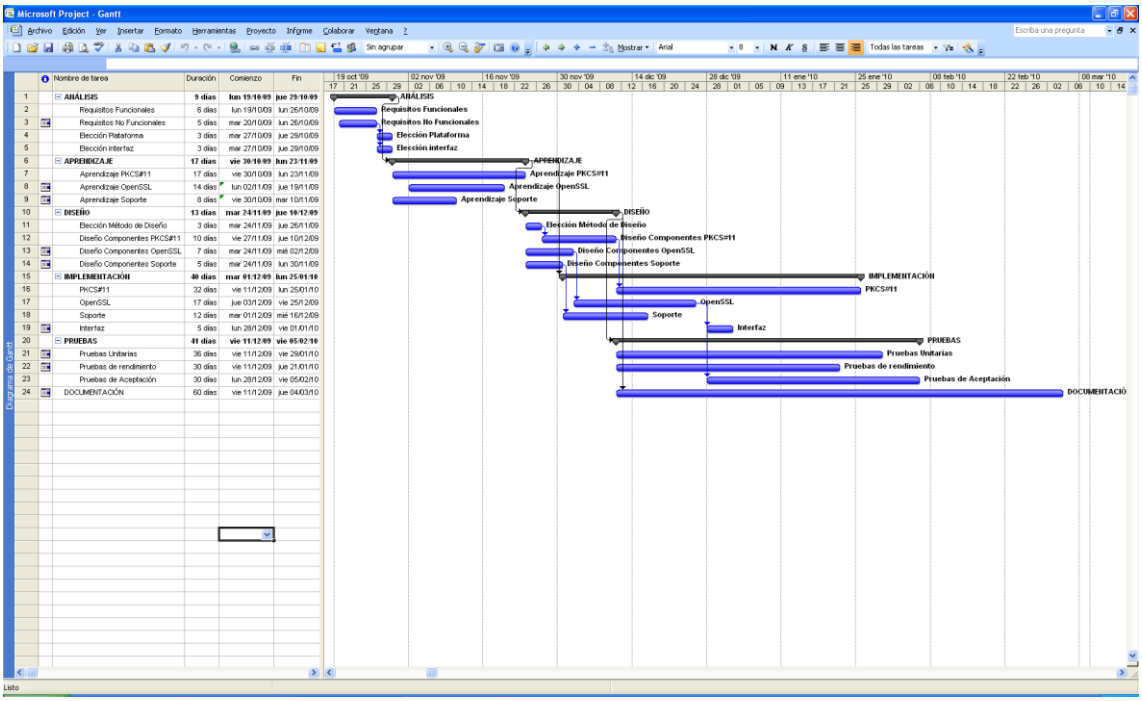

Ilustración 61: Pantalla de Project

Altova Umodel

Altova Umodel es un editor de diagramas UML con el que se obtienen buenos resultados desde el punto de vista visual. Además, permite representar fielmente en los diagramas que genera toda la semántica ofrecida por el lenguaje UML.

Este software ha sido utilizado para generar el diagrama de componentes y el de despliegue de la aplicación. La versión utilizada es la 2008.

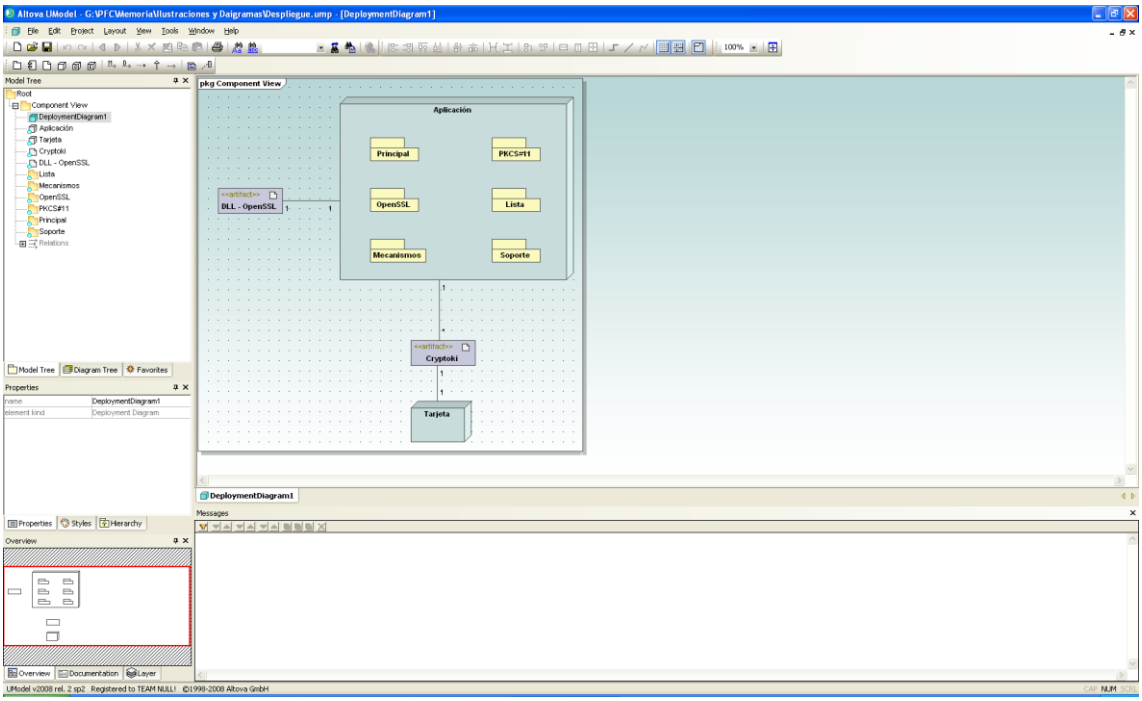

Ilustración 62: Pantalla de Altova Umodel

VMWare Player

VMWare Player es un software capaz de ejecutar máquinas virtuales y, a partir de su versión 3, también es capaz de generarlas. Una máquina virtual es una aplicación capaz de emular el hardware de una máquina física y ejecutar ésta dentro de un sistema operativo anfitrión.

VMWare Player es software gratuito y está disponible tanto para sistemas Unix como para Windows. Su arquitectura emula la arquitectura genérica de un PC (haciendo uso para ello del software físico de la máquina anfitriona). VMWare permite emular todo hardware genérico de un PC y, además, utilizar dispositivos externos conectados al PC anfitrión tales como USBs, lectores de tarjetas, impresoras, etc.

VMWare permite instalar y ejecutar básicamente Sistemas Unix (BSD, Solaris y Linux principalmente) y todos los sistemas Microsoft Windows.

La máquina virtual generada para el uso de Windows Server 2003 tenía las siguientes características: 512 MB de RAM, 2 núcleos (siempre se ejecutó en ordenadores físicos de 2 ó 4 cores), disco duro de 12 GB y red basada en NAT.

La versión de VMWare Player, utilizada en este proyecto para crear la máquina virtual del Windows Server 2003, es la 3.0.

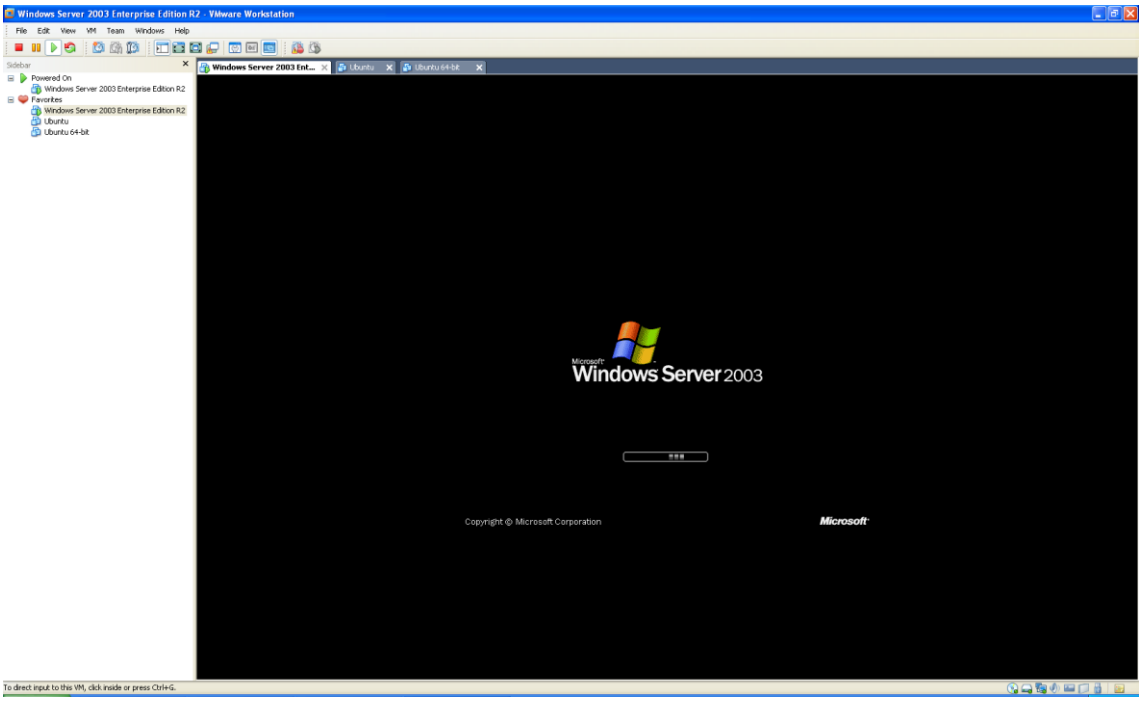

Ilustración 63: Pantalla de VMWare Player

ASN.1 Editor

ASN.1 Editor es un programa que permite visualizar de forma gráfica la estructura ASN.1 de cualquier fichero que la siga, como, por ejemplo, certificados digitales, ficheros PFX, etc.

ASN.1, además, permite ver la codificación DER o PEM de dichos ficheros y hacer conversiones de una a otra. En el caso de ficheros codificados en DER, permite resaltado de sintaxis para observar qué partes de la codificación se corresponden con las partes concretas del nodo físico. ASN.1 también permite la modificación de los nodos presentes en un fichero ASN.1.

Esta aplicación se utilizado para comprobar que los certificados y los CSR generados eran correctos a nivel ASN.1 ya que pequeños errores hacían que no fueran válidos.

Este es un programa gratuito para Windows y la versión utilizada fue la 1.0.20.

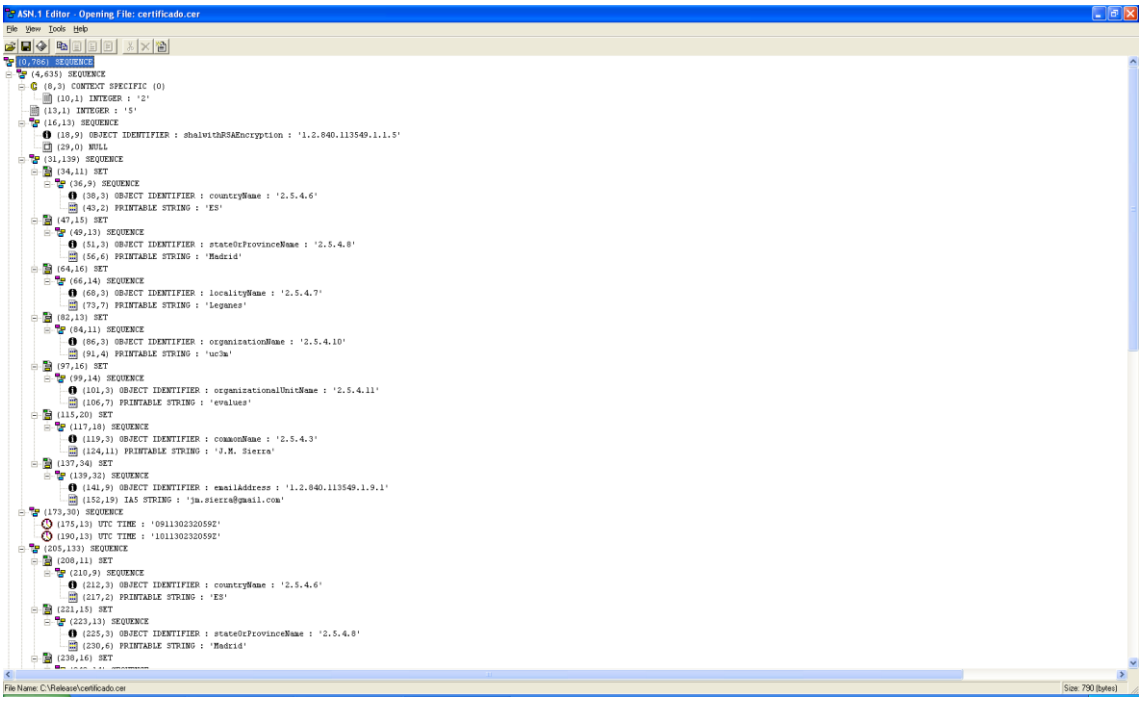

Ilustración 64: Pantalla de ASN.1 editor

Notepad++

Es un editor de textos especializado en la programación. Sus características son el resaltado de sintaxis y una potente herramienta de búsqueda de texto dentro de los ficheros fuente abiertos.

Notapad++ tiene gran cantidad de plugins que complementan su funcionalidad. De todos los plugins disponibles se ha usado el del visor Hexadecimal, que permite ver el texto en formato Hexadecimal (útil, por ejemplo, para ver las firmas digitales).

La versión de Notepad++ utilizada es la 5.6.8.

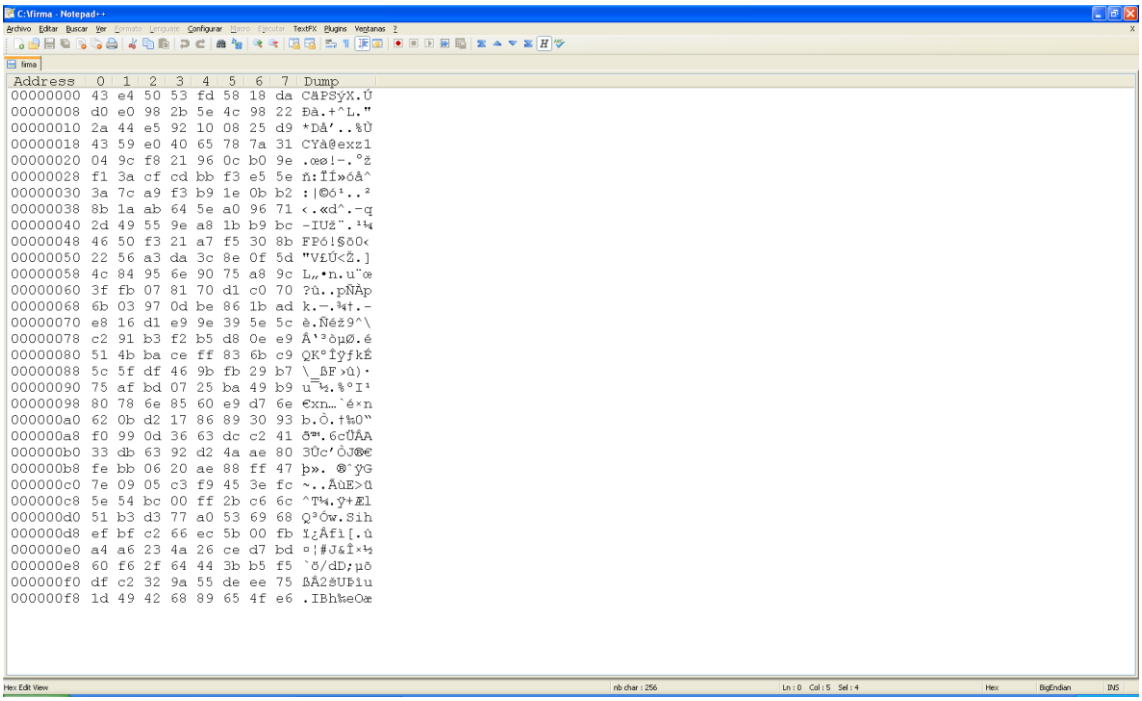

Ilustración 65: Pantalla de Notepad++

OPENSSL

Fichero de configuración

```
HOME            = .
RANDFILE        = $ENV::HOME/.rnd
oid_section     = new_oids
[ new_oids ]
[ ca ]
default_ca      = CA_default
dir             = ./CA
certs           = $dir/certs
crl_dir         = $dir/crl
database  = $dir/index.txt
new_certs_dir  = $dir/newcerts
certificate     = $dir/cacert.cer
serial          = $dir/serial
crlnumber = $dir/crlnumber
crl       = $dir/crl.pem
private_key     = $dir/private/cakey.pem
RANDFILE  = $dir/private/.rnd
name_opt  = ca_default
cert_opt  = ca_default
default_days   = 365
default_crl_days= 30
default_md      = sha1
preserve  = no
policy          = policy_match
[ policy_match ]
countryName             = optional
stateOrProvinceName = optional
localityName            = optional
organizationName        = optional
organizationalUnitName   = optional
commonName              = supplied
emailAddress            = optional
[ policy_anything ]
countryName             = optional
stateOrProvinceName = optional
localityName            = optional
organizationName        = optional
organizationalUnitName   = optional
commonName              = supplied
```

```
emailAddress        = optional
[ req ]
default_bits        = 2048
default_keyfile     = privkey.pem
distinguished_name  = req_distinguished_name
attributes          = req_attributes
x509_extensions     = v3_ca
[ req_distinguished_name ]
countryName         = Nombre del Pais (Codigo del pais)
countryName_default = ES
countryName_min     = 2
countryName_max     = 2
stateOrProvinceName = Provincia
stateOrProvinceName_default   = Madrid
localityName        = Localidad
0.organizationName  = Organziacion
0.organizationName_default    = UC3M
organizationalUnitName        = Departamento
organizationalUnitName_default     = Evalues
commonName          = Nombre
commonName_max      = 64
emailAddress        = Correo electronico
emailAddress_max    = 64
[ req_attributes ]
challengePassword   = A challenge password
challengePassword_min    = 4
challengePassword_max    = 20
unstructuredName    = An optional company name
[ usr_cert ]
basicConstraints=CA:FALSE
keyUsage       =     "digitalSignature",     "keyCertSign",
"nonRepudiation"
subjectKeyIdentifier     =hash
authorityKeyIdentifier   =keyid,issuer
[ v3_req ]
basicConstraints    = CA:FALSE
keyUsage      =      nonRepudiation,      digitalSignature,
keyEncipherment
[ v3_ca ]
subjectKeyIdentifier     =hash
authorityKeyIdentifier   =keyid:always,issuer:always
basicConstraints    = CA:true
[ crl_ext ]
authorityKeyIdentifier   =keyid:always,issuer:always
```

```
[ proxy_cert_ext ]
basicConstraints    =CA:FALSE
subjectKeyIdentifier     =hash
authorityKeyIdentifier   =keyid,issuer:always
proxyCertInfo=critical,language:id-ppl-
anyLanguage,pathlen:3,policy:foo
```

Comandos para gestionar OpenSSL

Crear una CA:

```
openssl req -new -x509 -extensions v3_ca -keyout <dir-
privado-conf>/cakey.pem -out cacert.pem -days <número-días>
-config openssl.cnf
```

Crear una petición de certificado:

```
openssl req -new -nodes -out <nombre-petición>.pem -config
openssl.cnf
```

Firmar una petición de certificado con una CA:

```
openssl ca -out <nombre-del-certificado-firmado>.pem -
config openssl.cnf -days <número-días> -infiles <nombde-
petición>.pem
```

Visualziar un certificado:

```
openssl x509 -in <nombre-certificado>.pem -noout -text
```

Convertir certificado de PEM a DER:

```
openssl x509 –in <certificado>.pem –inform PEM –out
<certificado-convertido>.der –outform DER
```

Convertir certificado de DER a PEM:

```
openssl x509 –in <certificado>.der –inform DER –out
<certificado-convertido>.pem –outform PEM
```

Convertir clave rsa de PEM a DER:

```
openssl rsa –in <clave>.pem –inform PEM –out <clave-
convertida>.der –outform DER
```

Convertir clave rsa de DER a PEM:

```
openssl rsa –in <clave>.der –inform DER –out <clave-
convertida>.pem –outform PEM
```

Exportar certificado y clave privada en formato PKCS#12:

```
openssl pkcs12 -export -out <pkcs#12-salida>.pfx -in
<fichero-con-certificado-y-clave-privada>.pem -name
"nombre-de-exportación"
```

Pasar de PKCS#12 a un fichero PEM que contiene el certificado y la clave:

```
openssl pkcs12 -in <pkcs#12>.pfx -out <certificado-y-clave-
de-salida>.pem -nodes
```

PKCS#11

Estructuras usadas de PKCS#11

Versión del Crytptoki:

```
typedef struct CK_VERSION {
     CK_BYTE major;
     CK_BYTE minor;
} CK_VERSION;
```

Información del Cryptoki:

```
typedef struct CK_INFO {
     CK_VERSION cryptokiVersion;
     CK_UTF8CHAR manufacturerID[32];
     CK_FLAGS flags;
     CK_UTF8CHAR libraryDescription[32];
     CK_VERSION libraryVersion;
} CK_INFO;
```

Información del Lector:

```
typedef struct CK_SLOT_INFO {
     CK_UTF8CHAR slotDescription[64];
     CK_UTF8CHAR manufacturerID[32];
     CK_FLAGS flags;
     CK_VERSION hardwareVersion;
     CK_VERSION firmwareVersion;
} CK_SLOT_INFO;
```

Información de la tarjeta:

```
typedef struct CK_TOKEN_INFO {
     CK_UTF8CHAR label[32];
     CK_UTF8CHAR manufacturerID[32];
     CK_UTF8CHAR model[16];
     CK_CHAR serialNumber[16];
     CK_FLAGS flags;
     CK_ULONG ulMaxSessionCount;
     CK_ULONG ulSessionCount;
     CK_ULONG ulMaxRwSessionCount;
     CK_ULONG ulRwSessionCount;
     CK_ULONG ulMaxPinLen;
     CK_ULONG ulMinPinLen;
     CK_ULONG ulTotalPublicMemory;
     CK_ULONG ulFreePublicMemory;
     CK_ULONG ulTotalPrivateMemory;
     CK_ULONG ulFreePrivateMemory;
```

```
     CK_VERSION hardwareVersion;
     CK_VERSION firmwareVersion;
     CK_CHAR utcTime[16];
} CK_TOKEN_INFO;
```

Información de la sesión:

```
typedef struct CK_SESSION_INFO {
     CK_SLOT_ID slotID;
     CK_STATE state;
     CK_FLAGS flags;
     CK_ULONG ulDeviceError;
} CK_SESSION_INFO;
```

Estructura de los atributos:

```
typedef struct CK_ATTRIBUTE {
     CK_ATTRIBUTE_TYPE type;
     CK_VOID_PTR pValue;
     CK_ULONG ulValueLen;
} CK_ATTRIBUTE;
```

Estructura de las fechas:

```
typedef struct CK_DATE {
     CK_CHAR year[4];
     CK_CHAR month[2];
     CK_CHAR day[2];
} CK_DATE;
```

Estructura de los mecanismos:

```
typedef struct CK_MECHANISM {
     CK_MECHANISM_TYPE mechanism;
     CK_VOID_PTR pParameter;
     CK_ULONG ulParameterLen;
} CK_MECHANISM;
```

Información de los mecanismos:

```
typedef struct CK_MECHANISM_INFO {
     CK_ULONG ulMinKeySize;
     CK_ULONG ulMaxKeySize;
     CK_FLAGS flags;
} CK_MECHANISM_INFO;
```

Flags de PKCS#11

Información del lector:

Nombre del Flag	Flag	Significado
CKF_TOKEN_PRESENT	0x00000001	Indica si hay token.
CKF_REMOVABLE_DEVICE	0x00000002	Indica si la es dispositivo es extraíble.
CKF_HW_SLOT	0x00000004	Indica si es una tarjeta hardware.

Tabla 41: Información del lector

Información de la tarjeta:

Nombre del Flag	Flag	Significado
CKF_RNG	0x00000001	Indica si el token posee generador de números aleatorios.
CKF_WRITE_PROTECTED	0x00000002	Indica si el token está protegido contra escritura
CKF_LOGIN_REQUIRED	0x00000004	Indica si se requiere el PIN para ejecutar alguna función.
CKF_USER_PIN_INITIALIZED	0x00000008	Indica si el PIN de usuario normal está inicializado.
CKF_RESTORE_KEY_NOT_NEEDED	0x00000020	Indica si se puede restaurar una sesión criptográfica.
CKF_CLOCK_ON_TOKEN	0x00000040	Indica si el token posee un reloj interno.
CKF_PROTECTED_AUTHENTICATION_PATH	0x00000100	Indica si el token tiene una ruta de autenticación segura.
CKF_DUAL_CRYPTO_OPERATIONS	0x00000200	Indica si se permite la criptografía dual.
CKF_TOKEN_INITIALIZED	0x00000400	Indica si el token ha sido inicializado (si tiene formato).
CKF_SECONDARY_AUTHENTICATION	0x00000800	Obsoleto
CKF_USER_PIN_COUNT_LOW	0x00010000	Indica si la última vez se introdujo un PIN erróneo.
CKF_USER_PIN_FINAL_TRY	0x00020000	Indica que un PIN incorrecto más bloqueará el token.
CKF_USER_PIN_LOCKED	0x00040000	Indica si el PIN está bloqueado.
CKF_USER_PIN_TO_BE_CHANGED	0x00080000	Indica si el PIN de la tarjeta es el PIN original.
CKF_SO_PIN_COUNT_LOW	0x00100000	Indica si el PIN del oficial de seguridad fue introducido mal la última vez.
CKF_SO_PIN_FINAL_TRY	0x00200000	Indica que un PIN incorrecto más bloqueará al oficial de seguridad.
CKF_SO_PIN_LOCKED	0x00400000	Indica si el PIN del oficial de seguridad está bloqueado.
CKF_SO_PIN_TO_BE_CHANGED	0x00800000	Indica si el PIN del oficial de

Nombre del Flag	Flag	Significado
		seguridad de la tarjeta es el PIN original.

Tabla 42: información de la tarjeta

Información de la sesión:

Nombre del Flag	Flag	Significado
CKF_RW_SESSION	0x00000002	Indica si es una sesión de Lectura/Escritura.
CKF_SERIAL_SESSION	0x00000004	Siempre debe ser TRUE.

Tabla 43: Información de la sesión

Información de los mecanismos:

Nombre del Flag	Flag	Significado
CKF_HW	0x00000001	Indica si el mecanismo se implementa por hardware.
CKF_ENCRYPT	0x00000100	Indica si se puede usar con C_Encrypt
CKF_DECRYPT	0x00000200	Indica si se puede usar con C_Decrypt
CKF_DIGEST	0x00000400	Indica si se puede usar con C_DigestInit
CKF_SIGN	0x00000800	Indica si se puede usar con C_SignInit
CKF_SIGN_RECOVER	0x00001000	Indica si se puede usar con C_SignRecoverinit
CKF_VERIFY	0x00002000	Indica si se puede usar con C_VerifyInit
CKF_VERIFY_RECOVER	0x00004000	Indica si se puede usar con C_VerifyRecoverInit
CKF_GENERATE	0x00008000	Indica si se puede usar con C_GeenrateKey
CKF_GENERATE_KEY_PAIR	0x00010000	Indica si se puede usar con C_GenerateKeyPair
CKF_WRAP	0x00020000	Indica si se puede usar con C_WrapKey
CKF_UNWRAP	0x00040000	Indica si se puede usar con C_UnwrapKey
CKF_DERIVE	0x00080000	Indica si se puede usar con C_DeriveKey
CKF_EXTENSION	0x80000000	Indica si hay una extensión para el flag.

Tabla 44: información de los mecanismos